

AN APPROACH TO A GENERAL PROGRAMMING LANGUAGE PROCESSOR THROUGH DECISION TABLES

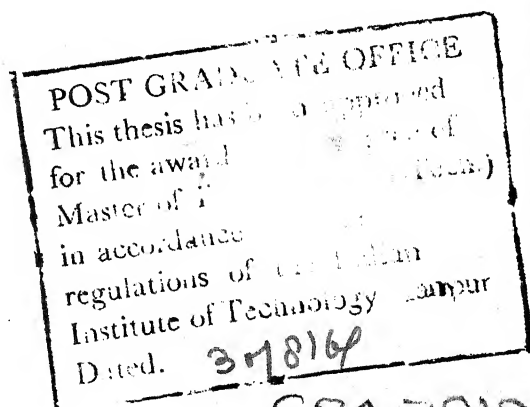
A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY IN ELECTRICAL ENGINEERING



BY
K. V. RAMANA RAO

to the



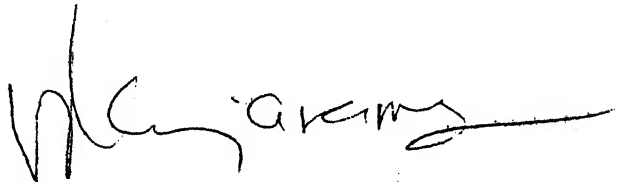
EE-1969-M-RAO-APP

621-3819
R 141a

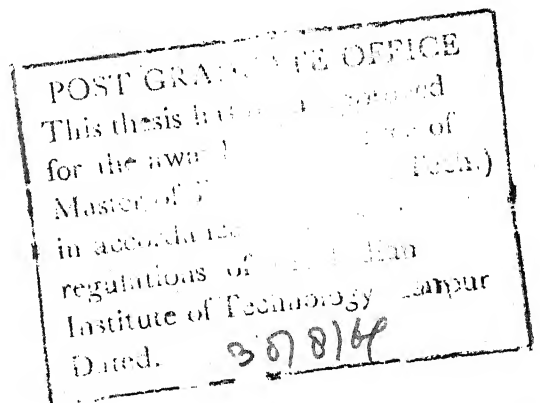
DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
AUGUST, 1969

CERTIFICATE

This is to certify that the thesis entitled
'An Approach to a General Programming Language
Processor through Decision Tables' is a record of
the work carried out under my supervision and that
it has not been submitted elsewhere for a degree.



Professor of Electrical Engineering
and Head, Computer Centre
Indian Institute of Technology
Kanpur.



ACKNOWLEDGEMENTS

It is with genuine pleasure that I express my deep sense of gratitude to my thesis advisor, Dr. V. Rajaraman for his counsel and encouragement throughout this work. I am grateful to my friend Mr. C.R. Muthukrishnan for several useful discussions. I thank the staff of the Computer Centre, I.I.T, Kanpur for the facilities provided on the IBM 7044 System.

K.V. Ramana Rao

ABSTRACT

The two main approaches towards a general programming language processor viz., the syntax-directed and the macro-processor approaches, have been critically examined. A new approach using decision tables for displaying syntax and decision tables in conjunction with a set of canonical actions for specifying semantics of programming languages has been presented. The problem of mechanical translation of syntax has been considered and a simple algorithm for translating syntax specification in the form of state graphs to decision tables has been suggested.

CONTENTS

	INTRODUCTION	1
CHAPTER I	: AN OVERVIEW OF SYNTAX-DIRECTED COMPILERS	
1.1	Introduction	5
1.2	Some typical attempts towards the syntax-directed compiler	6
1.2.1	Compiler-compiler of Brooker, Morris, et al.	6
1.2.2	Feldman's formal semantic language	7
1.2.3	Use of Post's canonical systems by Donovan and Ledgard	9
1.2.4	Reeves' parse-edit machine	12
1.3	Some remarks on the syntax-directed compiler	17
CHAPTER II	: THE MACRO-PROCESSOR APPROACH TO TRANSLATOR WRITING SYSTEMS	
2.1	Introduction	19
2.2	Earlier work on the macro-processor approach	19
2.2.1	Ferguson's meta-assembler	20
2.2.2	McIlroy's extendible compiler	21
2.2.3	Halpern's work on the macro- processor approach	22
2.3	Discussion of the macro-processor approach	24
2.3.1	Advantages	24
2.3.2	Drawbacks and possible remedies	27

CHAPTER III	:	DECISION TABLE AS A TOOL FOR EXPRESSING SYNTAX	
3.1		Introduction to decision tables	29
3.1.1		Structure and definitions	29
3.1.2		Illustration of a decision table	31
3.1.3		Some remarks on decision tables	32
3.2		Decision tables for syntax recognition	33
3.3		Practical illustration of the concept	41
CHAPTER IV	:	DECISION TABLE AS A TOOL FOR EXPRESSING SEMANTICS	
4.1		Gravity of the problem of semantics	44
4.2		Canonical actions for specifying semantics	46
4.3		Role of decision tables in specifying semantics	47
4.4		Suggested set of canonical actions	48
4.5		Desirability of direct translation of decision tables to order code	52
4.6		Example of semantics decision table	54
CHAPTER V	:	MECHANICAL TRANSLATION OF SYNTAX	
5.1		Introduction	57
5.1.1		Desirability of automatic translation of syntax	57
5.1.2		Drawbacks of direct translation of BNF	57
5.1.3		Use of an intermediate level in translating BNF	59

5.2	Earlier work on state graph approach to language description	60
5.2.1	Conway's separable transition diagram compiler	60
5.2.2	Gorn's summary of specification languages	63
5.3.1	Pragmatics of translating state graphs into machine representation	65
5.3.2	A simple algorithm to generate decision tables from an incidence matrix	66
5.3.3	Implementation of the above algorithm for binary machines	71
	CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH	74
	BIBLIOGRAPHY	77
APPENDIX A	: LISTING OF THE LITTLETRAN COMPILER	80

INTRODUCTION

The increase in the applications of computers has been phenomenal over the past decade and half. From numerical computation to analytical manipulation of mathematical expressions and from symbol manipulation to speech recognition, the spectrum of computer applications is enormously wide. Apart from their breadth, these applications, coming from numerous different problem environments, are very diverse in nature. The computer being a rigidly monolingual machine, the demand for new translators which can present a problem-oriented interface to the user (by way of programming languages) has been constantly increasing.

The writing of such a translator consumed several man-years in the early stages of computer evolution and persistent efforts have brought this down to several man-months. Nevertheless the gap in time between the conception of a programming language and its implementation has remained wide enough to discourage any experimentation on the former and at times render a concept stale by the time it is implemented. Besides, the complete machine-orientation of the conventional translator has not only

rendered itself useless for other machines, but it has also resulted in any attempt to modify it (to introduce new features in a programming language), becoming a Herculean task.

The realization of the aforesaid features has motivated several research workers to attempt a general programming language processor to which the description of a source language and its translation (into a given machine language) can be fed as input whereupon it acts as a compiler for that source language. The interest in this approach has been so great that the literature is replete with accounts of efforts in this direction. A considerable portion of these efforts have borrowed models from linguistics to describe programming languages. The inherent inadequacy of these models to provide a complete description of programming languages accompanied by the penchant of research workers for formalization has resulted in the general programming language processor being well above the comprehension of the average programmer.

In this thesis, we have attempted to present a pragmatic approach towards a general programming language processor, using decision tables. Decision tables have a simple tabular structure which assists in a precise and readable presentation of decision-making procedures. What

makes decision tables active documentation devices is their amenability to be translated directly into computer programs. Thus we contend that a general programming language processor using decision tables can be flexible, efficient and readable at the same time.

We now present a brief account of our approach. We have demonstrated the feasibility of displaying a given syntax specification in the form of decision tables. This not only facilitates readability and extendibility, but it also provides scope for an efficient machine implementation.

In any pure syntax-directed approach, specification of semantics becomes a bottleneck since there is no standard metalanguage for specifying semantics. We have circumvented this problem by using a set of canonical actions in conjunction with decision tables. These canonical actions represent the basic functions that a computer can perform and thus constitute the most natural form of specifying the semantics of programming languages.

To retain the advantages of a natural form of syntax specification and efficiency of machine implementation (which are conflicting in their requirements), we have suggested the conversion of the former into an intermediate form by hand. Choosing state graph as this intermediate

form, we have presented a simple algorithm which translates this into decision tables. By thus standardizing the specification of a translator in terms of decision tables, we felt that a common efficiency criterion for general programming language processors could be worked out.

For illustration purposes, we have programmed a small compiler for a limited subset of FORTRAN IV using decision tables embedded in FORTRAN IV (a translator for which is available at our computer centre). Recognition of syntactic units and specification of semantics have been displayed by decision tables. Feasibility being the main motivation, the effort lacks detail.

CHAPTER I

AN OVERVIEW OF SYNTAX-DIRECTED COMPILERS

1.1 INTRODUCTION

Ever since Irons' historic paper (IRONS61), the syntax-directed compiler has been the cynosure of research workers in the area of programming language processors. Numerous accounts of efforts towards this (so called) universal processor for programming languages have appeared in the literature. Feldman and Gries (FELD68) have made an exhaustive coverage of these efforts in a review paper. We feel that we cannot surpass them either in their experience or in their meticulous coverage of the work in this area. So we do not intend to duplicate their effort by attempting another state-of-the-art survey. However, since in this thesis, we suggest an approach towards a general programming language processor (which is not devoid of the concepts used in syntax-directed compilers), we present here four typical efforts in this area. We elaborate (somewhat critically) on two efforts in particular viz., the use of Post's canonical systems in (DON67) and the parse-edit machine in (REEV) since we feel that the first one has a sound formalism and the second one is a maiden suggestion to realize the syntax-directed compiler in the form of a special purpose computer.

1.2 SOME TYPICAL ATTEMPTS TOWARDS THE SYNTAX-DIRECTED COMPILER

1.2.1 COMPILER-COMPILER OF BROOKER, MORRIS, ET AL.

The compiler-compiler of Brooker and Morris ((BROOK67), (ROSE64)) is one of the earliest attempts made in this direction. In the notation of this system, a phrase is a string of elementary symbols or class identifiers; a phrase class is defined in a form similar to Backus-Naur Form (hereafter abbreviated as BNF). A format class is similar to a phrase class, with the distinction that each format in a format class has an associated format routine. The format routines and the associated source statement formats are the generators which actually control translation and other compiling operations. Certain basic statement formats for manipulation of numbers, temporary storage, indices and data structure at the translation level are built into the system; other auxillary statement formats may be written using the basic statements and previously defined auxillary statements. The format routines themselves are written in the language defined by the basic and auxillary statements.

A compiler is built up on this structure by adding format routines, each a list of statements in formats

already in the system, in a macro-building fashion. With enough source statement formats added, the system will act as a compiler for the language so described. The compiler-compiler has been successfully used to produce several compilers which are compared with their hand-coded counterparts in (BROOK67b).

1.2.2 FELDMAN'S FORMAL SEMANTIC LANGUAGE

Feldman (FELD66) has introduced a semantic metalanguage known as Formal Semantic Language (abbreviated hereafter as FSL) which he uses to automate the postsyn-tactic aspects of compiler-writing. The syntax is described in a production language, which is based on a recognizer with a single push-down stack. As each character is scanned, it is placed on top of the stack and the first few elements are scanned by the productions which describe the source language. The production statements are formatted in five fields - a) A label b) A "picture" of a stack configuration which is the pattern tested against the stack. The rest of the production is executed if a match occurs; otherwise the next production in sequence is tested c) A second (optional) "picture" of the stack configuration. If this field is non-empty and a match occurs in (b), the stack is transformed to this pattern at the end of execution of this production d) An indication of semantic action to

be taken: this may call for the execution of a semantic routine, an error indication or termination of compilation

e) The label of the next production to be compared to the stack, along with (optionally) a request for the next input character to go into the stack.

The semantic routines are written in the formalism FSL and are physically separated from the productions. An FSL program consists of two sections viz., a declaration part and the main body. The declaration part describes the storage to be used by the translator, which may include cells, constants, tables and stacks. The body of the program consists of semantic descriptions of individual constructs of the source language. The basic unit is a labelled statement; the label forms the link to the production language. A clear distinction is made between compile-time actions and run-time actions the latter being enclosed between 'CODE(' on the left and ')' on the right. The statement types of FSL include most of the common constructs of programming languages. A statement consists of a series of commands, each of which may be a) An assignment statement b) An operation of an element of the data structure (eg. table-lookup) c) A transfer of control d) A call on a system macro e) A conditional statement and almost any of the above in CODE brackets to indicate

run-time execution. The Formula ALGOL group at Carnegie Mellon University, Pittsburgh has made extensive use of FSL to produce several compilers (ITUR66).

1.2.3 USE OF POST'S CANONICAL SYSTEMS BY DONOVAN AND LEDGARD

Donovan and Ledgard (DON67) have presented a scheme for specifying the syntax and translation of computer languages using Post's canonical systems. Their scheme has evolved from the theory that canonic systems can be used to specify any recursively enumerable set. They have used canonic systems to define two examples of recursively enumerable sets viz., the set of syntactically legal programs comprising a computer language, and the set of ordered pairs specifying the translation of programs in one language to programs in another. We include here an example from their report to illustrate the scheme. A subset of PL/I was chosen by the authors for the definition of syntax and translation into IBM System/360 assembly language. The syntax of a "GO TO" statement is expressed as follows:

1 label | — GO TO 1; go to stm with ref label 1, label var \wedge

This canon can be informally read as :

If 1 is a member of the set "label", then "GO TO 1;"

is a member of the set "go to stm with ref label l and label var \wedge (null)".

To include the translation into 360 assembly language, the above canon is modified as :

$$1 \text{ label } \left| \begin{array}{l} \text{GO TO l; go to stm with ref label l, label} \\ \text{vars } \wedge \text{ assembler stms B } \quad 1 \quad * \text{ BRANCH TO l} \end{array} \right.$$

The authors contend that the proposed scheme can account for the contex-sensitive features of programming languages (eg. No two statement labels should be the same; No variable should appear in more than one declaration and so on) which phrase structure grammars and BNF cannot account for. We feel however that this point has not been given as much emphasis as it deserves. Canonic systems appear to be a good scheme with explicit provision for context-sensitive grammars and hence constitute a pragmatic formalism. The authors' statement that BNF coupled with "a few English sentences" provides a concise description of languages is truly amazing. Conciseness, though an undisputed virtue, should not be at the expense of relevant details. We do not share the authors' belief that canonic systems will not replace BNF to describe programming languages to "people". The feasibility of a formalism for describing programming languages that can

appeal to the lay user is very doubtful in the light of the numerous attempts made so far.

Canonic systems provide a precise and uncluttered way of describing programming languages. Though they incorporate semantics along with syntax, features like the presence of messages in the generated code ensure that readability is not impaired. However there seems to be some difficulty in implementing operator precedence. In the scheme suggested by the authors, addition and subtraction are carried out in separate registers from multiplication and division. Such a procedure would prove very inefficient on a machine with a single set of registers (Accumulator and Multiply Quotient) for arithmetic operations. In as much as this is an instant-code generation scheme, any code optimization can be attempted only at assembly level.

The scheme has not yet been implemented in full detail. Since this scheme with its intrinsic clarity and precision is likely to appeal to the formal linguist as well as the informal programmer, we feel that such an implementation will be worthwhile and will enable one to assess the true merits of the scheme.

1.2.4 REEVES' PARSE-EDIT MACHINE

Reeves (REEV67) has simulated a parse-edit machine in ALGOL on a KDF9 computer. The machine simulated is a special purpose stored program computer which acts as a syntax-directed translator. A program for this machine represents the syntax and semantics of a given source language. The machine reads as data, under control of its program, a text allegedly written in the source language. If the machine finds the source text syntactically legal, it produces an output determined by the semantics of the source language.

The author believes that "students and teachers of computer science may find it of value to have readily available facilities for practical experimentation in the definition and manipulation of formal languages". In the light of this belief, one would expect two features from this paper - a) A clean and precise formalism for expressing syntax and semantics, and b) A discussion of the pragmatics involved in the realization of the machine proposed. However, to one's dismay, one can find neither of these features in the elaborate presentation. This will be evident as we review some of the important aspects of the paper.

Though the author has retained BNF with minor modifications for expressing syntax, he insists on proving that this metalanguage is capable of describing itself. The result is a clumsy notation in which one has to spend considerable time in distinguishing between metasymbols and actual symbols in order to grasp the intent of the productions. We quote here one of the productions presented in Section 2 of the paper:

$$\langle \text{Variable} \rangle ::= \langle * \neq = \rangle \Rightarrow ;$$

This can be informally read as - "A variable is defined as a "<" followed by a sequence (including none or many) of symbols other than ">" followed by a ">". This painstaking exercise does not throw any new light on the metalanguage proposed. It has been realized in (DON67) that no metalanguage proposed hitherto is capable of expressing the syntax of any arbitrary source language. The author's desire to have "a more formal and compact definition" is another instance of formalization at the expense of elegance and readability.

In Section 3 of the paper, the author introduces some of the operation codes of the parser machine. Also provided is a tutorial exercise in defining an "unsigned integer" whose contribution towards understanding the parser machine can be seriously doubted. The parser machine

has two subroutine cue and link instructions, three conditional jump instructions, three input instructions and one output instruction. Two examples are worked out using these instructions and they are self-explanatory.

In Sections 4, 5 and 6, the author outlines a scheme for the automatic conversion of the metalanguage into the machine code of the parser machine. He introduces a number of edit codes which are to be suitably embedded in the metalanguage program describing the syntax of a source language. It is precisely these edit commands that introduce the semantics of the source language in its syntax specification. To accomplish the automatic translation of metalanguage, the parser machine is extended. The parser machine parses an input string of the source language and if it is syntactically legal, it produces an output which comprises source language symbols interspersed with edit codes. At this point, the editor machine takes over. It interprets the edit codes and takes suitable actions. The output of the editor machine will be the translation (in a target language) of the source language text. For the sake of clarity, we list out all the steps involved as follows :

- a) Input the specification (of the syntax and semantics) of the source language in the metalanguage to the extended parser machine.

- b) The extended parser machine produces an output in its machine code, which is in effect the compiler for the given source language.
- c) Input the program produced in (b) to the extended parser machine. The data to this program is a text written in the source language.
- d) The extended parser machine produces a symboloc output interspersed with edit commands.
- e) The editor machine (initiated by an EDIT W command) scans the output stream produced in (d) and executes all the edit commands. The resulting output is the translation (in a target language) of the source language text.

Sections 7 and 8 introduce some edit codes amidst a profusion of detail about the assembly language and its implementation. It is annoying to find these details jeopardize the sequence of the important aspects of the scheme presented. The edit codes presented here perform internal number conversion, name searching and bit selection. Sections 9 and 10 dwell upon some aspects of the simulator.

We sum up here our impressions about the scheme. As the structure of the machine to be used is not imposed a priori (as is the case in practice) there can be

considerable flexibility in the design of the formalism for describing the syntax and the semantics of source languages. We feel that the author has not exploited this significant feature. Consequently, as far as the user is concerned, the existence of a separate machine for syntax-directed translation does not in any way simplify his task of describing a desired language. One would compromise to such an inconvenience provided that the machine suggested is either amenable for easy implementation or if it supersedes other existing syntax-directed translators in terms of efficiency. One finds no comment on either of these aspects in the paper, though having written the simulator, the author is best equipped to venture it.

Though the author confesses at the outset that his account of the scheme is "rather detailed though deliberately elementary", the deliberation on some occasions far supersedes one's expectations and clouds the issue being discussed (vide Section 3). An orderly description of the parse and the edit machines followed by examples for subsets of well-known languages would have been more instructive and comprehensible than the meta-grammar presented in Appendix 2 of the paper. Also some of the details of assembly could be dispensed with since such techniques are well established and published.

1.3 SOME REMARKS ON THE SYNTAX-DIRECTED COMPILER

In spite of the tremendous energies expended on it, the syntax-directed compiler has failed to cross the boundaries of the research worker's cell. This can be traced back to some (avoidable) features that have characterized the work in this area. Firstly, most of the attempts have been pedantic rather than pragmatic in their approach, giving undue importance to formalization sometimes at the expense of detail. One cannot otherwise explain the persistent usage of models like the phrase structure grammar which have been proved to be inadequate for syntax expression (CHOM63). Secondly, little attention has been paid to the realization of algorithms for translating syntax and semantics to make the resulting compilers efficient. In the absence of this, the commercial world of computer manufacturers would be content with their efficient hand-coded compilers. Thirdly, most of the efforts have been isolated in nature. As a result, the ingenuity of one has not been exploited by others. Fourthly, there has not been the slightest attempt to standardize the formal notation being used. This may cause the switch-over of the user from one syntax-directed compiler to another to be so difficult that he may prefer to use an existing programming

language. Finally, as a compound result of the above features, the possibility of a common efficiency criterion for syntax-directed compilers is remote. The fact that there is no such well defined criterion for conventional compilers does not eliminate this problem.

CHAPTER II

THE MACRO-PROCESSOR APPROACH TO TRANSLATOR WRITING SYSTEMS

2.1 INTRODUCTION

The idea of a general processor for programming languages has been pursued by two schools of thought - one taking the syntax-directed approach and the other taking the macro-processor approach. It behoves us to confess here that we are not trying to present the case for the macro-processor approach (Halpern (HAL68) has already done this with considerable vehemence). We have attempted to present the highlights as well as the drawbacks of this approach as applied to the realization of a general processor for programming languages. Our intent in this presentation will become clear in later chapters where we suggest an approach which in effect is a combination of the syntax-directed and macro-processor approaches.

2.2 EARLIER WORK ON THE MACRO PROCESSOR APPROACH

Expositions of the use of macros as a tool for compiler-writing have appeared early in the literature. Bennett and Neuman (BENN64) discuss a scheme of extending compilers by using macros and string concatenation. The macros describing the new features to be added to the source

language are arranged at the beginning of the source deck. The authors illustrate this with macros which can extract a given portion of any machine word. It is suggested that this feature could be utilized in communicating between separate programs.

2.2.1 FERGUSON'S META-ASSEMBLER

Ferguson (FERG66) introduces the concept of meta-assemblers. This concept has arisen from the realization that all assemblers have many features in common. By building procedures for handling such things as symbol tables, location counters and macros, one could speed up the writing of particular assemblers. The input to the meta-assembler comprises the word size of the machine, internal number representation etc. The output of the meta-assembler would be an assembler for the given machine. Ferguson further says that the meta-assembler could be used in the role of a compiler also by providing certain additional features, since there is always an overlap between the facilities required by an assembler and a compiler - eg. Stacks, Table-lookup etc.. For instance consider the (source) statement :

```
IF F(A) PLUS 5 EQ G(B) GO TO L
```

For enabling the meta-assembler to handle this statement, Ferguson suggests a scheme called many-many macro in which

IF, PLUS, EQ and GO TO are defined as prefix operators. The many-many macro has features for using and updating state information during text replacement. It has potential applications in compiler-writing since it can handle a sentential form completely. One cannot comment on the efficiency of this scheme since Ferguson does not present any details of implementation.

2.2.2 MCILROY'S EXTENDIBLE COMPILER

McIlroy (MCIL60) is one of the early advocates who suggested macros as a wholesome tool for compiler-writing. He emphasizes that powerful compilers can be built from a small set of functions (primitives). He illustrates this for the case of algebraic translation. The main advantage claimed by him for macros is the "definitional extension" of a source language. The primary requirement for such an extension is that the scanner of the compiler be appended to accept new source language statements. McIlroy has worked out an example showing how this can be achieved for variable field scanning which is commonly used in compilers. His remark that statements effective at object time should have counterparts effective at compile-time is very significant. If this is realized, the interpretation of source statements could be implemented with ease and the problem of semantics is solved to a large extent.

2.2.3 HALPERN'S WORK ON THE MACRO-PROCESSOR APPROACH

Halpern (HAL68) is the most vociferous proponent of the macro-processor approach. We first mention here some features of the system XPOP (HAL64) which he has developed. In a language implemented on this system, any statement is a call on a macro written in IBM7090 assembly language. The system assumes the responsibility of recognizing the macro calls (eg. GO TO, IF, DO etc.) which occur in source statements. The punctuation of a source statement can be specified as a set of parameters in the macro by which it is to be recognized. This ensures flexibility in the format of source statements. A macro definition can specify noise words which are to be ignored and keywords which are to be retained. Thus the source language can have an English-like structure with a limited vocabulary. This is no small advantage for a lay user of the source language.

For the implementation of source languages, XPOP offers certain elegant facilities. It allows the compiler-writer to defer code generation from a part of the source program until some criteria (specified by the programmer) are satisfied. This is useful in compiling DO statements. Further the system has a number of trace and debugging aids which would be of immense help in checking the compiler.

Knowledge of the assembly language is indispensable for the user of XPOP. In this respect, XPOP resembles a conventional monolingual compiler. However the important difference is that the latter operates at a 'micro' level whereas XPOP operates at a 'macro' level. A good analogy would be the difference between electronic circuits made of discrete components and integrated circuits. In the former case, one has to always start at a basic level no matter how complex the desired circuit is. In the latter case, one is given a set of carefully chosen modules which can be combined to realize the desired circuit. Thus not only is the time spent on elaborate details at a basic level saved but the designer can aim at more complex and powerful circuits without the accompanying sacrifice of time and labour.

XPOP appears vulnerable to implement ALGOL-like languages which have a high degree of structure (as observed by Feldman and Gries (FELD68)). The nesting of macro calls would be quite involved in such cases. With the advent of third generation computers with hardware stack operations (eg. Burroughs B8500), this drawback may not be serious.

In a subsequent paper (HAL68), Halpern makes a vehement attack on the opponents of the macro-processor approach. He brings out several significant points regarding the incessant quest for a general processor by numerous

research workers. With clinical expertise, he dissects the syntax-directed compiler in its present form and shows that the fault lies in the inadequate formalism around which it is built viz., the phrase structure grammar. However in his anxiety to defend the cause of the macro-processor approach, he tries to cloud some of the inescapable deficiencies of this approach (viz., machine oriented notation, redundant coding etc.). A more pragmatic stand would have been to concede these drawbacks and suggest possible remedies. We venture to make this attempt in the next section.

2.3 DISCUSSION OF THE MACRO-PROCESSOR APPROACH

2.3.1 ADVANTAGES

We discuss here five important advantages of the macro-processor approach. We follow this up with a mention of its main disadvantages and possible means by which they could be obviated.

a) Uniformity of the modus operandi

The macro-processor approach offers a uniform technique for compiler-writing. It has evolved out of the belief that a set of basic functions (primitives) can be used to build powerful compilers. McIlroy (MCIL60) has stated this in one of the earliest papers on this approach. Without uniformity in approach, there can be no common

ground on which comparison can be made among several attempts. In such a case, the collective progress in the area would be extremely slow however good the individual attempts may be. This is made evident by the fact that the numerous diverse attempts towards the syntax-directed approach (which far outnumber those towards the macro-processor approach) have resulted in several good individual efforts which because of their isolated nature have produced very little interaction among their authors.

b) Natural method of handling semantics

In the syntax-directed approach, recognition of a syntactic unit is a streamlined process but its interpretation remains a blockhead. (Several individual attempts have been made to overcome this problem but have failed to yield a common satisfactory solution). The gravity of the problem of semantics is magnified in the case of languages for symbol manipulation. For instance in SNOBOL "STRING1 \bowtie STRING2 \bowtie = \bowtie STRING3" appears to be an innocuous statement whose syntax could be easily expressed in BNF but its interpretation in English would run into several lines (and many more in machine code). The primitives used in the macro-processor approach represent "actions" which the machine can perform. Thus they constitute the most natural tool for expressing the semantics of programming languages.

c) Extendibility of compilers

In order to meet the demand of new applications, programming languages must have the capacity to evolve with time. The macro-processor approach offers ample scope for extending compilers. New actions could be defined in terms of the existing macros or new macros defined if necessary.

d) Possibility of approaching natural language

As demonstrated in Halpern's XPOP (HAL64), restrictions on the format of source language statements could be considerably reduced with the aid of the macro-processor approach. Thus the source language could be brought nearer to natural language. This has a far reaching effect on improving man-machine communication. Now that computer time-sharing techniques are well established, the user has direct access to the machine. This "nearness" could be better exploited if the language of communication resembles natural language at least in a limited sense. The computer would be more welcome with a child's vocabulary rather than with an unnatural and formal adult vocabulary by the lay user.

e) Efficiency of implementation

The efficiency of the macro processor is largely dependent on how the assembler handles the macros and their

c) Extendibility of compilers

In order to meet the demand of new applications, programming languages must have the capacity to evolve with time. The macro-processor approach offers ample scope for extending compilers. New actions could be defined in terms of the existing macros or new macros defined if necessary.

d) Possibility of approaching natural language

As demonstrated in Halpern's XPOP (HAL64), restrictions on the format of source language statements could be considerably reduced with the aid of the macro-processor approach. Thus the source language could be brought nearer to natural language. This has a far reaching effect on improving man-machine communication. Now that computer time-sharing techniques are well established, the user has direct access to the machine. This "nearness" could be better exploited if the language of communication resembles natural language at least in a limited sense. The computer would be more welcome with a child's vocabulary rather than with an unnatural and formal adult vocabulary by the lay user.

e) Efficiency of implementation

The efficiency of the macro processor is largely dependent on how the assembler handles the macros and their

nesting. Techniques to do this are well established and hence it is possible to have a common efficiency criterion for macro-processors. This would take us one step towards the elusive efficiency criterion for compilers.

2.3.2 DRAWBACKS AND POSSIBLE REMEDIES

a) Object program inefficiency

A macro is an open subroutine as a result of which repeated calls on it would result in the generation of redundant coding. This leads to inefficient object programs. This is the main objection of the antagonists of the macro-processor approach. We feel that this problem is not serious any more in the light of the following reasons.

Recent trends in computer design tend to bestow on the user the liberty of building his own operation codes rather than give him an inflexible set of (standard) operation codes. Thus microprogramming has found its place in third generation computers. This being the case, the macro can be looked upon as a powerful operation code the implementation of which could be carried out at a micro level, rendering its execution much faster. Then repeated calls on a macro would be synonymous with the repeated execution of an operation code in a conventional computer. Further hardware operated stacks (mentioned in 2.2.3) would facilitate the nesting of macros.

b) Machine dependence

Macros are made up of the operation codes of a given machine and hence any system built around macros would be machine dependent. This is the other important objection raised against the macro-processor approach. The answer to this problem, we feel, lies in choosing a sufficiently large standard set of actions(which are in effect macro calls). Having agreed upon such a standard set, it only requires these actions to be coded ONCE for a given machine. We hasten to add that this does not solve the problem of machine dependence in its entirety. The question always arises as to what should be done if the user requires new primitives. They have to be coded in machine language. We state (somewhat philosophically) that it is not perhaps possible to totally eliminate machine dependency from a compiler. However the solution suggested by us, we hope, reduces machine dependency to a considerable extent.

In conclusion, we contend that with the present state of computer technology, macros can be purged of their drawbacks while at the same time retaining their advantages.

CHAPTER III

DECISION TABLE AS A TOOL FOR EXPRESSING SYNTAX

3.1 INTRODUCTION TO DECISION TABLES

3.1.1 STRUCTURE AND DEFINITIONS

Decision tables are a method of specifying

a) the relevant conditions to be met in a given situation
and b) the set of actions to be taken when these conditions
are satisfied. The name 'Decision Table' has arisen since
a decision-making procedure is displayed in the form of a
table. An excellent introduction to decision tables is
given in (KING67). Therefore we will only define the terms
necessary for our discussion.

	Rule 1		Rule n
Condition Stub 1	Condition		Condition
	Entry 11		Entry 1n
.....			
Condition Stub m	Condition		Condition
	Entry m1		Entry mn
.....			
Action Stub 1	Action		Action
	Entry 11		Entry 1n
.....			
Action Stub i	Action		Action
	Entry i1		Entry in

Figure 3.1 Structure of a Decision Table.

Figure 3.1 shows the structure of a decision table. A decision table has four quadrants. The upper left quadrant contains 'Condition Stubs' which specify the conditions to be met. The upper right quadrant contains 'Condition Entries' which specify the truth or falsity of the condition stub in that row. The lower left quadrant contains 'Action Stubs' which are the actions to be taken. The lower right quadrant contains 'Action Entries' which specify whether the action stub in that row is relevant or not. Each column in the combination of the right quadrants is called a 'Rule'. The significance of a rule is that when the conditions relevant in that column are satisfied, the actions denoted by the action entries are obeyed sequentially.

Decision tables are of three types - a) Limited Entry b) Extended Entry and c) Mixed Entry. In limited entry decision tables, the condition entries can be Y(Yes), N(No) or - (Don't care) and the action entries for the actions pertinent to a rule are denoted by X. In extended entry decision tables, a part of the condition or the action can extend into the corresponding entry. In mixed entry decision tables, both of the above features are permitted the restriction being that a given row must contain only one type (limited or extended) of entry.

3.1.2 ILLUSTRATION OF A DECISION TABLE

For illustration purposes, we state a procedure for selecting a team of cricket players and display the procedure in the form of a decision table. If a player is a batsman and a fielder; or a bowler and a fielder; or a wicket-keeper; or a batsman and a bowler; or a batsman, bowler and fielder; or a batsman and wicket-keeper, he is enrolled into the team. In all other cases, he is rejected.

	R1	R2	R3	R4	R5	R6	ELSE
Batsman	Y	-	N	Y	Y	Y	
Bowler	N	Y	-	Y	Y	-	
Fielder	Y	Y	-	Y	N	-	
Wicket-keeper	-	-	Y	N	-	Y	
Enrolled in the team	X	X	X	X	X	X	
Rejected							X

Figure 3.2 Example of a Decision Table.

The procedure just mentioned is shown in the form of a limited entry decision table in Figure 3.2. Since there are 4 condition stubs, a maximum of 2^4 i.e. 16 rules is possible. However since only 6 combinations of the

conditions are pertinent to the procedure, the ELSE rule specifies that a player is rejected under all other conditions.

3.1.3 SOME REMARKS ON DECISION TABLES

Decision tables provide a precise, concise and (yet) readable description of a procedure. They assist completeness of specification by making conspicuous any omission in the logical sequence of a problem definition. They pinpoint ambiguities in problem specification and thus serve as a useful diagnostic aid. They encourage modularity by making it possible to divide a complex procedure into a set of linked decision tables. They serve as a good documentation device. The fact that decision tables can be inputted to a computer directly makes them an active documentation device rather than a passive one like flowcharts.

Though not an explicit disadvantage, one precaution must be observed regarding the usage of decision tables. Decision tables have a parallel structure (the order of testing the rules and conditions being immaterial). Hence forcing decision tables on a procedure in which the order of testing the conditions is important could be awkward.

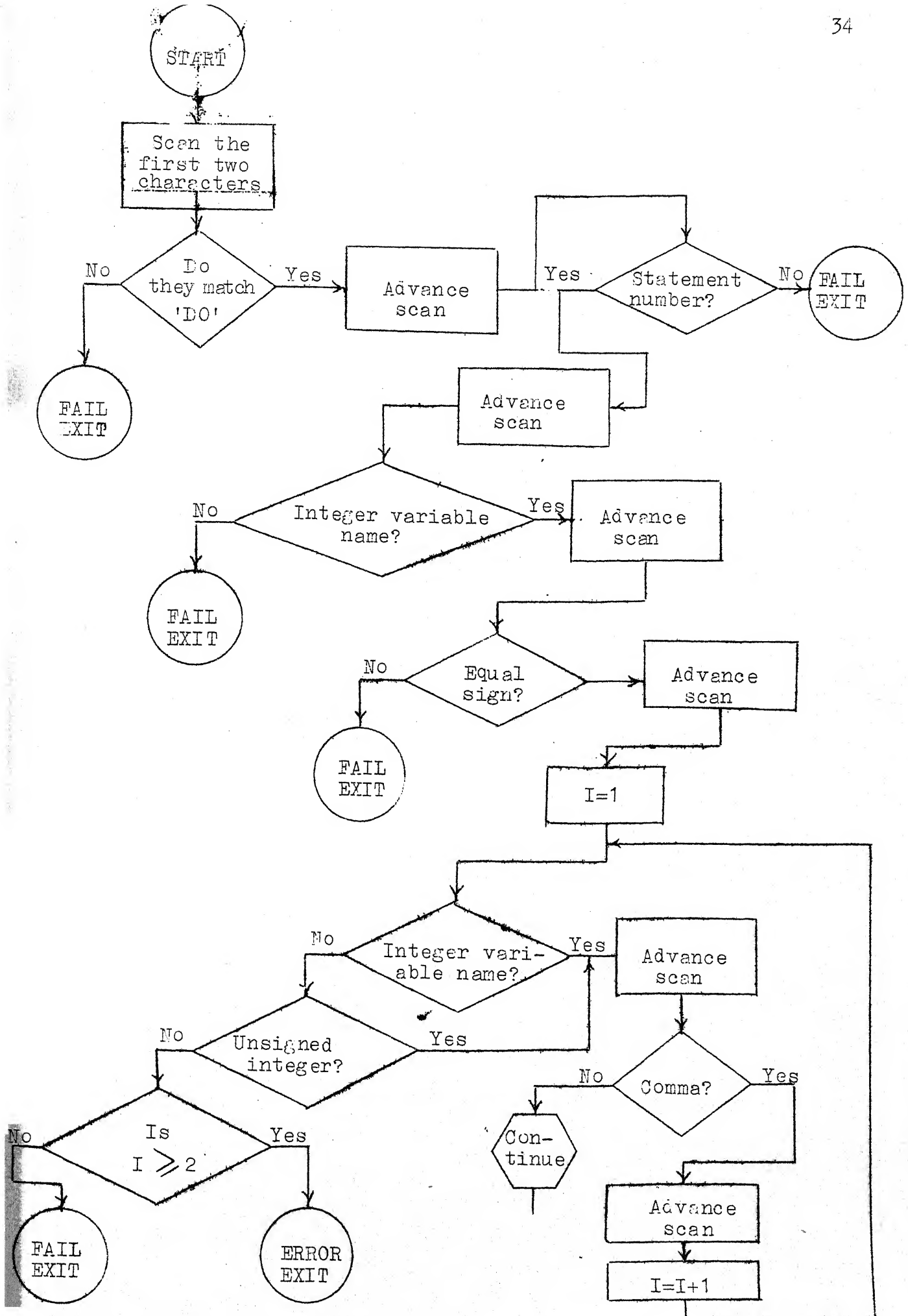
3.2 DECISION TABLES FOR SYNTAX RECOGNITION

We contend here that decision tables constitute a convenient tool for syntax recognition. Any syntax recognizing process is a multi-decision procedure. For instance let us consider a typical BNF production depicting the syntax of a DO statement.

$$\begin{aligned}
 \langle \text{Unlabelled DO statement} \rangle ::= & \text{DO} \langle \text{Statement Number} \rangle \langle \text{Integer} \\
 & \text{Variable Name} \rangle = \{ \langle \text{Unsigned Integer} \rangle | \\
 & \langle \text{Integer Variable Name} \rangle \} , \\
 & \{ \langle \text{Unsigned Integer} \rangle | \langle \text{Integer} \\
 & \text{Variable Name} \rangle \} [\{ , \{ \langle \text{Unsigned} \\
 & \text{Integer} \rangle | \langle \text{Integer Variable Name} \rangle \}]]
 \end{aligned}$$

Given this syntax, to check whether a source language statement is an unlabelled DO statement or not, the procedure to be adopted by a recognizer is shown in the form of a flowchart in Figure 3.3. The flowchart has been used to avoid the lengthy and cumbersome natural language interpretation. It can be seen that a number of conditions are to be satisfied and actions taken depending on the former's outcome. The same procedure is shown as a set of linked decision tables in Figure 3.4.

The intent of the 10 decision-making boxes in the flow chart of Figure 3.3 is displayed by the 6 linked decision



(Continued on next page)

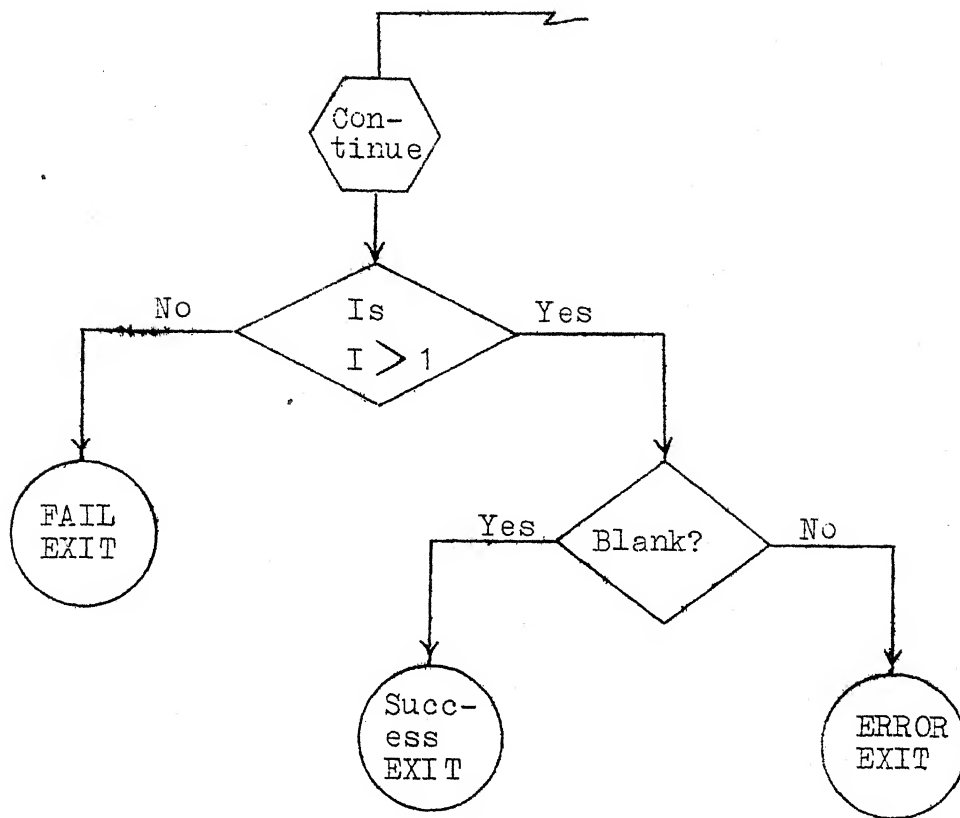


Figure 3.3. Flow Chart for recognizing the Syntax of a DO Statement.

Initialize Scan

Advance Scan

Table 1

ELSE

Scan (1)	= D	
Scan (2)	= 0	
Advance Scan	X	
Go to Table 2	X	
Fail exit		X

Table 2

Statement Number	Y	N
Advance Scan	X	
Go to Table 3	X	
Fail Exit		X

Table 3

Integer Variable Name	Y	N
Advance Scan	X	
Go to Table 4	X	
Fail Exit		X

(Continued on next page)

Table 4

Equal Sign	Y	N
Advance Scan	X	
I = 1	X	
Go to Table 5	X	
Fail Exit		X

Table 5

Integer Variable Name	Y	-	N	N
Unsigned Integer	-	Y	N	N
$I \geq 2$	-	-	N	Y
Advance Scan	X	X		
Go to Table 6	X	X		
Error Exit				X
Fail Exit			X	

(Continued on next page)

Table 6

Comma	Y	N	N	N
$I > 1$	-	N	Y	Y
Blank	-	-	Y	N
Advance Scan	X			
$I = I + 1$	X			
Go to Table 5	X			
Success Exit			X	
Fail Exit		X		
Error Exit				X

Figure 3.4 Linked Decision Tables for recognizing the Syntax of a DO statement.

tables of Figure 3.4. It may be seen that the first 4 decision tables have only one or two conditions. This is because they appear in a path which is strictly sequential. Unless one condition is satisfied, the next condition need not be tested. We have not contrived to combine these 4 tables since it would result in a clumsy and unnatural notation. Only when both branches of a decision-making box in a flowchart extend at least beyond one level, the combination of such boxes into a decision table would be meaningful. In constructing decision tables from a decision-making procedure which is partly sequential and partly parallel, the natural form of the procedure should be preserved. Only then can the resulting decision tables retain all the advantages of preciseness, conciseness, modularity and readability. Added to this is the main advantage that these decision tables can be directly fed to a computer.

The advantages of using decision tables for syntax recognition are two-fold: a) syntax recognition in compilers is a predictive process. It is assumed that a given source language statement belongs to a certain class of syntactic unit. If the source statement does not belong to this class, it will be evident during the early stage of syntactic analysis (for most syntactic units). However if the empirical

form of the source statement does confirm to a particular syntactic unit but certain parts of the source statement seem out of place, it means that the source statement is syntactically illegal. In such a case, suitable diagnostics are to be provided to the user so that he can locate and correct the error. By using decision tables for syntax recognition, we believe, it is possible to provide excellent error diagnostics. In a decision table depicting a given syntactic unit, the rules specify the combinations of conditions which recognize a legal syntactic unit. One could incorporate extra rules specifying some invalid combinations of conditions the action part being an error exit after printing diagnostics. These extra rules no doubt depend on the ingenuity of the programmers and his capacity to anticipate possible syntactic errors. Even if one cannot predict all such errors in advance, more rules can be added after gaining experience with the compiler. This is quite simple since decision tables are extremely amenable to modification. For pathological cases of syntactic errors, decision tables may not be able to provide suitable diagnostics and in our experience even hand-coded compilers fare no better in such cases.

b) A syntax recognizer using decision tables is efficient and it can be evaluated by an objective criterion. This is due to the fact that there exist several algorithms to

convert decision tables into computer programs for minimal storage (MUTHU69) or minimal time (GAN69) requirements. By using a minimal run-time algorithm, the syntax recognizer could be made fast (storage not being a stringent restriction except for in-core compilers). This is no small advantage considering the fact that the recognizer is a substantial part of a compiler and hence it will be used over and over again.

3.3 PRACTICAL ILLUSTRATION OF THE CONCEPT

To illustrate the feasibility of the concept suggested by us, we have developed a small compiler for a limited subset of FORTRAN IV. Illustration being the main motivation, the effort lacks detail and it is not expected to present a working language. These limitations resulted from the limited time at our disposal.

Use has been made of the Decision Table to FORTRAN IV translator DELTA available at our Computer Centre. Recognition of the syntactic units has been completely done with decision tables and auxillary details like scanning and character recognition have been programmed in FORTRAN IV.

We now present an example of a decision table which recognizes FORTRAN IV identifiers. This is shown in Figure 3.5. We explain here the parameters used in this

decision table. N is the number of characters in the identifier, which is initialized to 0 before entering the table. IFLAG1 and IFLAG2 are flags used by the routines checking for alphameric characters and digits respectively. These flags are initialized to 2 before entering the table and are set to 0 if the check fails and 1 if it succeeds. TEMP is an array of 6 words in which the characters of the identifier are stored one per word as and when they are recognized. C is the current position of the scan. The routine NUMBER checks for numbers; SCAN advances the scan every time it is called; CHAR checks for alphameric characters; and DIGIT checks for digits. With this background, the modus operandi of this decision table should be self-explanatory.

In conclusion we contend that decision tables comprise an efficient and pragmatic tool for syntax recognition and provide a convenient form of documentation.

10 CONTINUE

N	.EQ.0	.EQ.0	.GE.1	.GE.0	.GE.1	.EQ.6	.EQ.6	.LE.6	.EQ.0	.EQ.0
IFLAG1	.EQ.2	.EQ.0	.EQ.0	.EQ.1		.EQ.1		.EQ.0	.EQ.0	.EQ.0
IFLAG2	.EQ.2	.EQ.2		.EQ.1		.EQ.1	.EQ.1	.EQ.0	.EQ.1	.EQ.0
N	.LT.6	.LT.6	.LT.6					.GE.1		
CALL NUMBER	.N.	.N.	.N.	.N.	.N.	.N.	.N.	.N.	.X.	.N.
N=N+1	.N.	.N.	.N.	.X.	.X.	.X.	.X.	.N.	.N.	.N.
CALL SCAN	.N.	.N.	.N.	.X.	.X.	.X.	.X.	.N.	.N.	.N.
IFLAG2=2	.N.	.N.	.N.	.X.	.X.	.N.	.N.	.N.	.N.	.N.
CALL CHAR	.X.	.N.	.N.	.X.	.X.	.X.	.X.	.N.	.N.	.N.
CALL DIGIT	.N.	.X.	.X.	.N.	.N.	.X.	.X.	.N.	.N.	.N.
PRINT 15	.N.	.N.	.N.	.N.	.N.	.X.	.X.	.N.	.N.	.N.
IFLAG3=1	.N.	.N.	.N.	.N.	.N.	.N.	.N.	.N.	.N.	.N.
GO TO 10	.X.	.X.	.X.	.X.	.X.	.X.	.X.	.N.	.N.	.N.
RETURN	.N.	.N.	.N.	.N.	.N.	.N.	.N.	.N.	.N.	.X.

15 FORMAT(1X,* ERROR- MORE THAN SIX CHARACTERS IN A VARIABLE NAME*)

Figure 3.5 Decision Table for recognizing FORTRAN Identifiers.

CHAPTER IV

DECISION TABLE AS A TOOL FOR EXPRESSING SEMANTICS

4.1 GRAVITY OF THE PROBLEM OF SEMANTICS

Most of the work towards a general programming language processor has been centred around the syntax-directed approach which borrows heavily from the theory of natural languages (also known as linguistics). The phrase structure grammar is the backbone of this approach. The fallacy of adopting phrase structure grammar as the central model has been expounded in (HAL68). While the phrase structure grammar has been proved (CHOM63) to be an inadequate model for expressing syntax itself (being unable to account for any context-sensitive features of a language), its failure to provide for any specification of semantics is not surprising. It is more appropriate to say that it was never intended to account for any semantics. A grammar is a methodology for generating and/or recognizing syntactically legal sentences in a language. However advocates of the syntax-directed approach have contrived to extend the phrase structure grammar somehow to provide for the semantics of the programming language being defined. The result is an unnatural and cluttered notation which is difficult to

comprehend and more difficult to append.

Before we proceed to discuss a less popular but a more pragmatic approach, it is important that we appreciate the gravity of the problem of semantics with regard to programming languages. Natural languages have a standard form and though they are legion, they are not likely to multiply themselves in future. Therefore dictionaries have solved the problem of semantics for them. In contrast, programming languages are evolutionary by nature and they have no meaning by themselves. They HAVE to be translated into some machine language so that they give rise to executable programs. The requirements of computer users are so diverse that no single language can hope to meet these requirements. Besides, computers are produced by different manufacturers in a competitive atmosphere so that any possibility of a standard machine language is ruled out. Thus neither the source language nor the target language is standard. And the problem is to find a general scheme of semantics which can interpret a given source language in a given object language ! To add to the complexity of this already complex problem, new languages have come up for picture processing, natural language discourse etc. These languages describe activities to perform which the computer was not originally designed.

(We do acknowledge existence of exceptions which are special purpose computers like ILLIAC IV (KUCK 68)). So for each source statement in such languages, it will require a number of lines of machine code to execute it. With this picture in mind, we proceed to a discussion of a plausible solution.

4.2 CANONICAL ACTIONS FOR SPECIFYING SEMANTICS

As outlined in 2.3.1, we believe that the answer to the problem of semantics lies in the macro-processor approach. The intent of a source statement has to be executed by a machine. Since machine instructions exist at a micro level, it would be too cumbersome and time-consuming to interpret a source statement directly at machine language level. A good compromise would be to describe the capabilities of a machine through a set of canonical actions coded as macros. Three distinct advantages would emerge out of such an approach - 1) It is possible to incorporate run-time diagnostics into these macros. For instance, checking for overflows and underflows in arithmetic operations and checking for wild transfers within the memory can be carried out within these macros 2) Any machine dependent code optimization can be introduced within the macros. 3) By executing all user program instructions under the control of these macros, it is possible to implement interpreters.

An essential prerequisite is that the set of canonical actions should not be a closed one. (Otherwise we will be left with the same inflexibility encountered in conventional compilers). The user must have the facility to define new actions either by combining the existing canonical actions or at the machine level itself. This would leave scope for the ingenious user who wants to add his own features to an existing language. Possible remedies for the deficiencies of the macro-processor approach were discussed in 2.3.2 which justifies our having chosen the macro-processor approach in principle for specifying semantics.

4.3 RULE OF DECISION TABLES IN SPECIFYING SEMANTICS

We have now at our disposal a set of actions with which we can interpret source language statements. But the main question now is as to what actions are to be used under what conditions. For instance in FORTRAN, a DO statement has an optional specification of increment. If the increment is missing, it is to be taken as 1. This fact is not made use of by the syntax recognizer. It is the semantics routines which should recognize this fact and take suitable action. Thus a distinction is made between the syntax-checking and code-generating aspects of a compiler.

The main advantage accruing out of this clear distinction is that execution-time diagnostics for user programs can be incorporated within the canonical actions.

The decision-making to be done by semantics routines can be conveniently displayed by decision tables. This facilitates (in the form of additional rules) the checking of several features which cannot be checked by a syntax recognizer (since they do not come under its purview). For instance, in a DO statement of the form "DO nn ii = d1,d2,d3" the syntax recognizer does not check whether $d2 > d1$ or not. (If this condition is not met, most FORTRAN compilers realize it only after going through the DO loop once). We have already discussed in Chapter III, the advantages of using decision tables and hence we do not reiterate them here.

4.4 SUGGESTED SET OF CANONICAL ACTIONS

a) Arithmetic Operations

- | | |
|---------------------------|---|
| i) ADD A, B, RESULT | The values represented by the symbolic names A and B are added and the result stored in RESULT |
| ii) SUB A, B, RESULT | The value represented by the symbolic name B is subtracted from that represented by A and the result stored in RESULT |

specified by LABEL. If this field contains a number, it is interpreted as a FORTRAN statement number

ii) NJMP A, LABEL

If the value represented by A is negative, transfer is made to the statement with LABEL in its label field. Otherwise the next sequential instruction is executed

iii) PJMP A, LABEL

The interpretation is same as that of (ii) except that A should be positive

iv) ZJMP A, LABEL

The interpretation for this is same as that of (ii) except that A should be zero.

c) Stack Operations

i) STACK NAME

A stack is created and is given the name specified by NAME

ii) PUCH A, NAME

The contents of the first field are push down the stack specified by NAME

iii) POP SAVE, NAME

The top element of the stack specified by NAME is popped out from the stack and stored in SAVE

iv) TOP SAVE, NAME

The top element of the stack specified by NAME is stored in SAVE. The stack is unchanged.

d) Scan Operations

i) INPUT

A card containing a source statement is read and its contents are placed in an array BUFF. All blanks in the input statement are removed

ii) SCAN

This generates a pointer which initially points to the first word of the array BUFF. Each execution of SCAN advances this pointer by one position

iii) COPY ADRES

All the words in the BUFF area up to the one having the pointer are copied into the array ADRES.

e) Miscellaneous Operations

i) FLAG NAME, ON/OFF

The word with the symbolic address NAME is treated as a Boolean variable and it is set to 0 or 1 depending on whether the second field contains OFF or ON respectively

- ii) PRINT MESSAGE The message defined by MESSAGE
is printed on the system output
unit
- iii) COMPAR A,B,BOOL The contents of A and B are
compared and if a match is found,
the flag with the name BOOL is
turned ON
- iv) UPDATE A, B The address represented by A is
incremented by 1 and stored in B.

We do not claim that this is the most efficient or complete set of canonical actions. We would only say that this is a representative (or typical) set of actions and we did use them to write some decision tables for specifying semantics. To obtain a good working set of actions, it should be chosen by a group of experienced compiler-writers.

4.5 DESIRABILITY OF DIRECT TRANSLATION OF DECISION TABLES TO ORDER CODE

The small compiler which we have programmed (vide 2.4) was written in FORTRAN IV with embedded decision tables. So the actions in these decision tables are, of necessity, FORTRAN IV statements. This implies that the efficiency of the compiler written depends on the efficiency of the FORTRAN IV compiler which ultimately converts the source

program into the order code of the machine. To avoid such dependency, we feel it desirable to have a decision table to order code translator. In this connection, we should mention the algorithms available for converting decision tables to computer programs. There are two main approaches viz., the rule mask approach (KING 66) and the tree approach (REIN 66). In (MUTHU 69) the relative merits of these approaches have been discussed and a modified rule mask technique has been proposed. This new technique has one salient feature which was lacking in earlier ones viz., provision for incorporating excellent diagnostics at run-time like detection of ambiguities. Considering decision tables as a high level procedure oriented language, their use would be very restricted without proper diagnostics. Since separate algorithms have been presented for binary and BCD machines in (MUTHU69), it is possible to program a decision table to order code translator. To enhance the use of such a translator for writing syntax and semantics decision tables, we suggest that it should have two modes of working - GENERATE and EXECUTE. The actions encountered after a GENERATE command should be copied on to a tape. Finally this tape will contain the translation of the source program. The actions encountered after an EXECUTE command should be executed straight away. This gives the facility to make some tests at compile-time. Arbitrary

switching from one mode to another should be permissible. This, we hope, would constitute a flexible and efficient translator writing system.

4.6 EXAMPLE OF A SEMANTICS DECISION TABLE

We present here an example of a decision table using the canonical actions of 4.4 to specify the semantics of an arithmetic expression (vide Figure 4.1). The method used for conversion has been implemented in our small compiler a listing of which can be found in Appendix A.. This method may be called a modified Reverse Polish scheme. There are two stacks viz., MAIN and WORKING. In a pure reverse Polish scheme, the complete arithmetic expression is converted into reverse Polish notation and left in the MAIN stack. (The WORKING stack is used for delaying the pushing of operators into the MAIN stack until precedence relationship is satisfied). In the present scheme, POLISH is a decision table which examines the precedence relationship. When an operator is pushed into the MAIN stack (which contains the operands), control is handed over to the decision table ARITH. In this table, the top three elements of the MAIN stack are popped off (one operator + two operands). Code is generated depending on the operator and the result is stored in a temporary location. The address of this temporary location is pushed into MAIN and control is

handed back to POLISH. The process is terminated by POLISH when it encounters a blank (i.e. terminal character). POLISH examines operands as they are pushed into MAIN and turns on one of the two flags INTG or REAL according to FORTRAN conventions for variable names. If an expression contains both integer and real variable names, both flags will be on. In such a case, ARITH prints an error message and sets the mode to REAL. The function of the table should now be self-explanatory.

In conclusion, we contend that decision tables in conjunction with a set of canonical actions provide a plausible solution to the problem of semantics.

CHAPTER V

MECHANICAL TRANSLATION OF SYNTAX

5.1 INTRODUCTION

5.1.1 DESIRABILITY OF AUTOMATIC TRANSLATION OF SYNTAX

If the syntax-directed compiler is to be used by nonprofessional programmers for defining new languages, the syntax specification of source languages should be permitted through standard formalisms as far as possible. Varied and complex schemes of specification tend to discourage the user from attempting the task of defining a new language. Further, modifying the syntax specification to affect changes in the language would also become difficult. Since BNF is the most commonly accepted method of specifying syntax of programming languages, it would appear desirable to mechanize the translation of BNF into a suitable internal machine representation. Early (EARL 65) has done some work in this direction, details of which are not available.

5.1.2 DRAWBACKS OF DIRECT TRANSLATION OF BNF

We feel that it is not advisable to translate directly from BNF for the following reasons :

- a) BNF has no provision to accommodate the context-sensitive features of programming languages. (This has

been discussed in detail elsewhere (DON67) and hence it is not repeated here). It turns out that most programming languages do have context-sensitive features in their syntax and various methods have been adopted to circumvent this problem (eg. the explanations given under the caption "semantics" in the ALGOL 60 report). Viewed in this sense, BNF is not a "complete" scheme of syntax specification and hence its direct translation would only be a partial solution to the problem of specifying syntax.

b) If BNF is retained as such for specifying syntax, it will not be possible to incorporate appropriate diagnostics for syntactically erroneous source statements. The syntax-directed compiler can never hope to compete with the conventional hand-coded compiler, unless it provides reasonable diagnostics. It would not be appropriate to expect the user to submit error-free source programs all the time. We feel that this point has not received much attention in the literature.

c) Cantor (CANT62) has proved that there exists no algorithm which can decide whether a given Backus system (i.e. a set of productions in BNF) is ambiguous or not. So even if BNF were to be directly translated, the ambiguities in the given syntax specification would remain undetected. Thus the compiler itself will not be free of "bugs".

d) No algorithms, to our knowledge, have been reported in the literature for the automatic translation of BNF. Since the efficiency of the recognizer would depend directly on the outcome of this translation, this would take us farther from the elusive common efficiency criterion for syntax-directed compilers.

5.1.3 USE OF AN INTERMEDIATE LEVEL IN TRANSLATING BNF

In view of the arguments presented in 5.1.2, it is clear that the automatic translation of a BNF specification of syntax does not result in a complete recognizer. As an alternative, one could translate BNF into an intermediate form, the latter being chosen in such a way that it can overcome the deficiencies mentioned in the earlier section. Obviously this phase of translation (where "translation" is not used in a strict sense since the process involves translation + inclusion of certain features like context sensitivity) requires human attention. The price paid in this does not seem too heavy in the light of the inadequacies of a direct translation. The translation from this intermediate form into machine representation can certainly be automated.

The state graph, used to represent finite state machines in Automata Theory seems to be a useful intermediate form. Instances of the use of the state graphs

for syntax specification can be found in (GORN61) and (CON63) which are discussed in subsequent sections. We call the state graph specification as an intermediate form, since syntax specification has to necessarily originate at a natural level like BNF. The problem of conversion from this intermediate form will be discussed in 5.3 .

5.2 EARLIER WORK ON STATE GRAPH APPROACH TO LANGUAGE DESCRIPTION

5.2.1 CONWAY'S SEPARABLE TRANSITION DIAGRAM COMPILER

Conway (CON63) gives an account of a separable transition diagram compiler for COBOL. The scheme suggested by him is essentially centred around the state graph approach to the description of programming languages. The main virtue he claims for his compiler is its "separability". Depending on the storage capacity of a given computer, the compiler can be broken into a number of segments under the restrictions that a segment leaving the high speed core storage is irrevocable and that two working tapes are required. The first restriction is concerned with the modus operandi of the compiler and its feasibility is proved informally by the author with the aid of the "coroutine" concept. The second restriction can be easily met with by most computer installations.

Conway's compiler can be broadly divided into three phases - 1) Lexical Analysis 2) Syntactical Analysis and 3) Code Generation. During Lexical analysis, all the metavariables in the source text (eg. class identifiers in ALGOL) are reduced to single character codes. Now the modified input string is fed to a "diagrammer" which has a "window" to hold the source (or modified source) symbol. This serves as the input symbol for the starting state of the transition diagram (or state graph) being traversed currently. (Each syntactic type has a unique transition diagram which may also refer to other syntactic types). Thus the diagrammer acts as a top-down analyzer. The output of the diagrammer is an intermediate code (Suffix Polish in the author's example). At this point the code generator takes over and produces the object program.

The transition diagrams used in syntactical analysis are in essence borrowed from Automata Theory. The only difference is that an "action" is associated with every transition from one state to another. The author mentions two conditions on the transition diagrams which we intend to discuss in detail. The first one is the "No-loop condition" which says that no transition diagram will make a reference to itself without having first read an input symbol after it was entered. This ensures the

prevention of endless loops. The second one called the "No-Backup condition" defines away any need to specify an order in which the nonblank paths leading from a node are to be traversed. When a path has a syntactic type on it, an input symbol is called an initial input symbol if and only if it is in the input window and when the transition diagram defining the syntactic type on the path in question is entered, that symbol will be read before either a dead end occurs or the diagram is exited. Thus the No-Backup condition implies the No-Lpop condition and further states that for every node in the system of transition diagrams, the sets of initial input symbols of all the paths leading from that node are disjoint. The author cites three distinct advantages of the No-Backup condition viz.,

- 1) No backing up of the input string during scanning is necessary when a parse fails.
- 2) Specific diagnostics can be provided for syntactically erroneous source texts and
- 3) The ambiguity problem (CANT62) is legislated out of existence in languages defined by transition diagrams.

The first and the third advantages would be ideal for any compiler and (may be hence) certainly do not accrue together from the author's scheme. For instance when two syntactic units have one or more common units in the same order in an alternative, going monotonically along one path

may leave one at a dead end. The author suggests a multiple-exit transition diagram wherein different syntactic types with the common feature mentioned above result in different exits. When this type of modified transition diagram is used, there is no guarantee that the actions associated with several syntactic types are very different which means that the third advantage is lost. Thus the logical "Exclusive OR" of the first and third advantages can be considered the resultant advantage. Though the author is not totally unaware of these drawbacks, he would have done well to claim these advantages in a limited sense.

The author mentions at the outset that his scheme is a rebuttal to the belief that syntax-directed compilers are inefficient with regard to storage requirement and speed of translation. One of the main advantages claimed for syntax-directed compilers is the relative ease with which one could modify the source language. It is surprising that the author does not even mention schemes for making modifications to his separable transition diagram compiler.

5.2.2 GORN'S SUMMARY OF SPECIFICATION LANGUAGES

Gorn (GORN61) has summarized the well known specification languages for mechanical languages and their processors. Most of the specification languages are only

of theoretical interest since no significant attempt has been made to implement them on a computer. The notable exceptions are BNF and flow charts (the latter serving only as a documentation device rather than for machine implementation) which have gained popularity. Gorn's presentation is graphic and informative and we have nothing to add to it. Our interest is to borrow his concept of "Incidence Matrix" which will be used in 5.3.1 with some modifications. Gorn's idea was to develop the path and connectivity matrices from the (two) symbol - transition - state matrices (or incidence matrices). So he suggested a two-stage process in which the state graph is first converted into a symbol-state graph (a set of two directed graphs) and then developing from it the symbol-transition-state matrices. We have changed this into a one stage process by altering the structure of the incidence matrix. Our motivation, no doubt, was machine implementation. In the modified form of the incidence matrix, each column has a name representing a transition of the form XXX/YYY . The rows represent the states of the state graph. Retaining Gorn's notation for making entries in the incidence matrix, the process of developing the incidence matrix from the state graph is quite simple as will be shown in the next section.

5.3.1 PRAGMATICS OF TRANSLATING STATE GRAPHS INTO MACHINE REPRESENTATION

Having decided upon the state graph as an intermediate form of syntax representation, the immediate problem is its conversion into internal machine representation. It must be noted here that the result of translation of a state graph will be a part of a compiler which will be used for compiling many a source program. So the importance of the efficiency of the translated state graph (i.e. in effect the compiler) cannot be overemphasized. Since no rigorous algorithms to convert state graphs to computer programs have been published hitherto (to the best of our knowledge), the efficiency of such a conversion cannot be determined. It is advisable to convert state graphs into a suitable form for whose machine translation rigorous algorithms exist. We choose decision tables as the second intermediate form for the following reasons. The similarity between decision tables and finite state machines has been proved in (MUTHU69). Hence the translation from a state graph into a decision table is a natural rather than a contrived process. Secondly, there exist proven algorithms for converting decision tables into computer programs (eg. (MUTHU69)). Eventhough the translation of syntax from BNF to machine-usable form seems to be a multistage process, it has to be gone through only ONCE for a given source

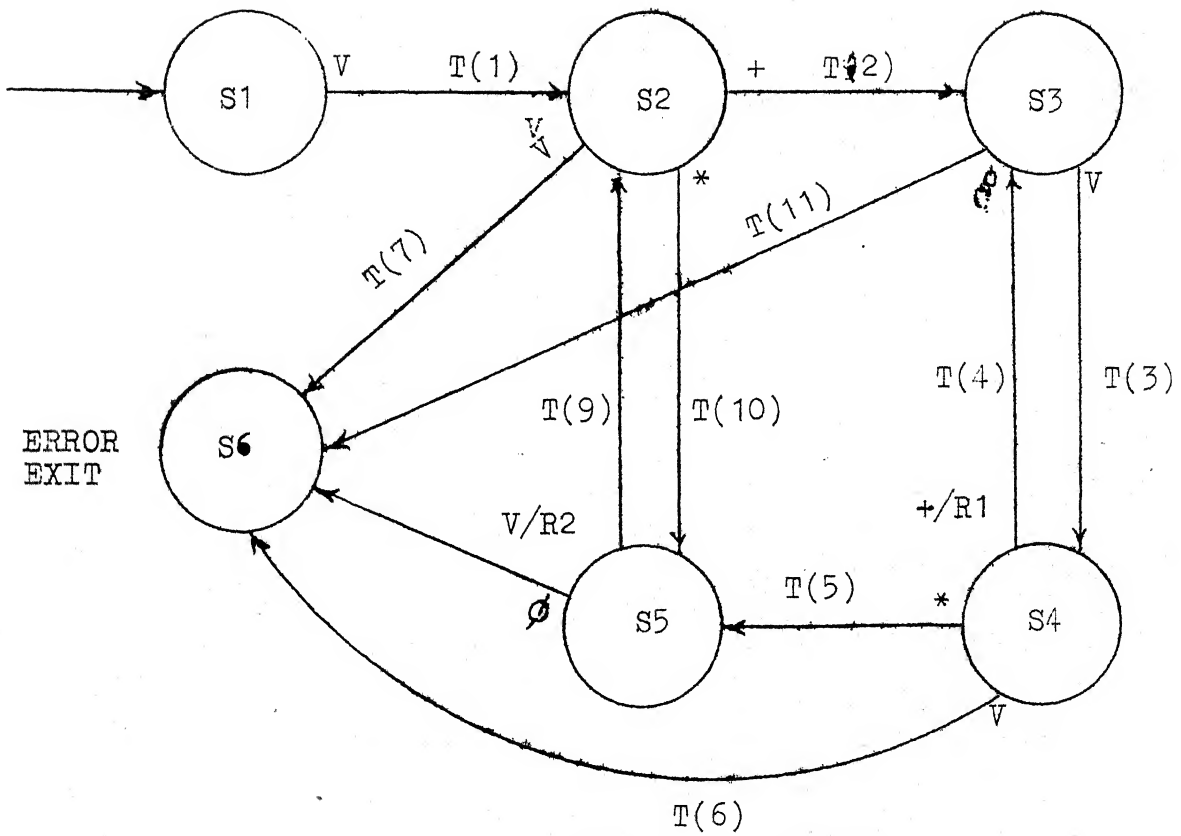


Figure 5.1 State graph for recognizing the Syntax in (a).

Whenever there is a / after such symbol, the entry therein indicates the action to be taken.

c) Incidence Matrix for the above State Graph

The transitions in the state graph are denoted by $T(1)$, $T(2)$... etc. The incidence matrix is developed by entering a -1 for an arrow (i.e. transition) leaving a state, $+1$ for an arrow entering it and ± 1 for an arrow leaving and entering the same state. When a transition is not pertinent to a given state, a 0 entry is made. The incidence matrix for the state graph in Figure 5.1 is shown in Figure 5.2.

State \ Transition											
	T(1)	T(2)	T(3)	T(4)	T(5)	T(6)	T(7)	T(8)	T(9)	T(10)	T(11)
S1	-1	0	0	0	0	0	0	0	0	0	0
S2	+1	-1	0	0	0	0	-1	0	+1	-1	0
S3	0	+1	-1	+1	0	0	0	0	0	0	-1
S4	0	0	+1	-1	-1	-1	0	0	0	0	0
S5	0	0	0	0	+1	0	0	-1	-1	+1	0
S6	0	0	0	0	0	+1	+1	+1	0	0	+1

Figure 5.2 Incidence Matrix for the State Graph in Figure 5.1.

T Array :

T(1) : V T(2) : + T(3) : V T(4) : +/R1
 T(5) : * T(6) : V T(7) : V T(8) : 0
 T(9) : V/R2 T(10) : * T(11) : 0

d) Conversion from Incidence Matrix to Decision Table

In the incidence matrix, each element of the T array will be of the form XXX/YYY where /YYY is optional. The format of the decision table to be generated is shown in Figure 5.3 informally.

Present State	i
Present Symbol	XXX
<hr/>	
Present State =	k
Reduction	R1 or R2
Exit	E or \emptyset

Figure 5.3 Format of the decision table to be generated.

Let the number of states be m and the number of transitions n . Then the incidence matrix will be $m \times n$ in size. Let c (which will be used for referring to the columns of the incidence matrix) be initialized to 1.

Step I

In the c^{th} column, if the i^{th} row contains -1 and the k^{th} row contains $+1$, make i as the condition entry

against the condition stub "Present State" and k as the action entry against the action stub "Present State = ". If k is equal j to j, where j is the number of the error state (6 in this example), put a marker "E" as the action entry against the action stub "Exit".

Step II

For the column number c of the incidence matrix access the cth element of the T array.

Step III

From the T array element (of the form XXX/YYY) take XXX and make it the condition entry against the condition stub "Present Symbol". If YYY is present, then make it the action entry against the action stub "Reduction".

Step IV

One rule of the decision table has been created. Increment c by 1 and if $c \leq n$ go to Step I. Otherwise halt. (The procedure is repeated for all the n columns of the incidence matrix).

Applying the above procedure for the incidence matrix in (c) the decision table in Figure 5.4 is obtained. For the sake of brevity, we have omitted details like reentering the table, moving the scan etc. which would be valid action entries for all rules without error exits.

The decision tables generated thus can be fed to a pre-processor (using any of the algorithms in (MUTHU69) or (GAN69)) which would convert them into the order code of the computer.

Present State	1	2	3	4	4	4	2	5	5	2	3
Present Symbol	V	+	V	+	*	V	V	0	V	*	0
Present State =	2	3	4	3	5	6	6	6	2	5	6
Reduction				R1				R2			
Exit						E	E	E			E

Figure 5.4 Decision table generated by the algorithm in (d).

5.3.3 IMPLEMENTATION OF THE ABOVE ALGORITHM FOR BINARY MACHINES

We present here a scheme for implementing the algorithm of 5.3.2 on binary machines. The entries in the incidence matrix viz., 0, -1, +1 and ± 1 are coded as 0, 1, 2 and 4 respectively. (The internal representation in the machine would be 000,001,010 and 100). We show an example of an incidence matrix with the above coding in Figure 5.5.

1	0	0	1	2
0	0	2	0	0
2	0	0	2	1
0	1	0	0	0
0	0	1	0	0
0	2	0	0	0

Figure 5.5. Example of a coded Incidence Matrix.

The coded incidence matrix should be fed columnwise to the computer. The problem is to find the state from which the transition originates and the state on which it terminates in each column (i.e. for a given transition). This can be done as follows. Consider the machine representation of a given column shown below :

001000010000000000 (Column 1 of Figure 5.5)

This should be ANDed logically with

001001001001001001 .

The result would be 001000000000000000. By finding the position of the 1 bit in this number (which can be accomplished by shifting the bits until a 1 is found and at the same time keeping track of the number of shifts) and dividing it by 3, we obtain the number of the state from which the transition originates.

Similarly by ANDing the column logically with 010010010010010010, locating the position of 1 in the resulting number and dividing it by 3, we can obtain the number of the state on which the transition terminates. In the event that a given transition originates and terminates on the same state, there would be only one entry (viz., 100) in the corresponding column of the incidence matrix. This possibility (which arises only if the earlier two have failed) can be detected by ANDing the column logically with 100100100100100100 and dividing by 3 the position of 1 in the resulting number. Since a byte in a binary machine can accommodate two octal digits, the columns of the incidence matrix can be conveniently packed into single machine words.

CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH

The syntax-directed approach towards a general programming language processor has concentrated mostly on the formalization of syntax leaving the problem of semantics in the background. Little has been said about the efficiency of translation of the syntax and the semantics of programming languages. In this thesis, we have shown that decision tables constitute a convenient tool for expressing syntax. Modification of the existing syntax of a programming language and incorporation of error diagnostics is simplified by using decision tables.

As outlined in (FEILD68), a syntax-directed approach, is appropriate only when the 'form' of the input to a program contains a significant portion of the 'content'. This limitation is overcome in the macro-processor approach which offers a powerful tool for describing the 'content' in terms of actions that a machine is capable of performing. We have used this feature in conjunction with decision tables to express the semantics of programming languages. The choice of optimal storage or optimal run-time for programs generated from decision tables is a significant advantage.

The main limitation on the usage of decision tables arises when the procedure to be displayed has sequential tests of conditions. Decision tables have a parallel structure. Hence any attempt to display a sequential procedure through decision tables is bound to be artificial and clumsy, crippling the main advantages of using them. Thus it is advisable to embed decision tables in a computer language which can take care of the sequential parts of a procedure.

As explained in Chapter V, direct translation of BNF does not appear to be a useful proposition. Though its conversion to state graph form calls for human intervention, the prospect of efficient translation compensates this to some extent. Since the incidence matrix is coded in binary form and decision tables are internally coded as binary vectors during translation, the possibility of a one-to-one translation could be explored.

Symbol manipulating languages provide a real touchstone for testing any general programming language processor. Hence it would be worthwhile to test the approach suggested by us by implementing a symbol manipulating language like SNOBOL. We could not attempt this in the limited time at our disposal.

Object code optimization is regaining prominence. It would be interesting to try decision tables for displaying under what conditions code optimization could be carried out. Since this is bound to cost in time, it may be incorporated as an optional postsemantic optimizer.

BIBLIOGRAPHY

- (BENN64) Bennett, R.K. and Neumann, D.H. Extension of existing compilers by sophisticated use of macros. Comm.ACM. Vol.7 (1964) p 541
- (BROOK67) Brooker, R.A., Morris, D., and Rohl, J.S. Compiler-compiler facilities in Atlas autocode. Comput. J. Vol.9 (1967) p 350
- (BROOK67b) Brooker, R.A., Morris, D., and Rohl, J.S. Experience with the compiler-compiler. Comput. J. Vol.9 (1967) p 345
- (CANT 62) Cantor, D.G. On the ambiguity problem of Backus systems. J. ACM. Vol.9 (1962) p 477
- (CHOM63) Chomsky, N. Formal properties of grammars. In Handbook of Mathematical Psychology. Vol.2 John Wiley & Sons, Inc. (1963) p 323
- (CON63) Conway, M.E. Design of a separable transition-diagram compiler. Comm.ACM. Vol.6 (1963) p 396
- (DON67) Donovan, J.J. and Ledgard, H.F. Canonic systems and their application to programming languages. Memo. MAC-M-347 (1967) Project MAC-Massachusetts Institute of Technology, Cambridge
- (EARL65) Earley, J.C. Generating a recognizer for a BNF grammar. Computation Center Rep. (1965) Carnegie Institute of Technology, Pittsburgh
- (FELD66) Feldman, J.A. A formal semantics for computer languages and its application in a compiler-compiler. Comm.ACM. Vol.9 (1966) p 3
- (FELD68) Feldman, J.A. and Gries, D. Translator writing systems. Comm.ACM. Vol.11 (1968) p 77
- (FERG66) Ferguson, D.E. Evolution of the meta-assembly program. Comm.ACM. Vol.9 (1966) p 190

- (GAN69) Ganapathy, S. Information theory applied to decision tables. Master's Thesis (1969) Indian Institute of Technology, Kanpur.
- (GORN61) Gorn, S. Specification languages for mechanical languages and their processors, a baker's dozen. Comm.ACM. Vol.4 (1961) p 532
- (HAL64) Halpern, M.I. XPOP: a metalanguage without metaphysics. Proc. AFIPS 1964 FJCC Vol.26 (1964) p 57
- (HAL68) Halpern, M.I. Toward a general processor for programming languages. Comm.ACM. Vol.11 (1968) p 15
- (IRONS61) Irons, E.T. A syntax directed compiler for ALGOL 60. Comm.ACM. Vol.4 (1961) p 51
- (ITUR66) Iturriaga, R., Standish, T.A., Krutar, R.A. and Earley, J.C. Techniques and advantages of using the formal compiler writing system ESL to implement a formula ALGOL compiler. Proc. AFIPS 1966 SJCC Vol.28 (1966) p 241
- (KING66) King, P.J.H. Conversion of decision tables to computer programs by rule mask techniques. Comm.ACM. Vol.9 (1966) p 796
- (KING67) King, P.J.H. Decision tables. Comput.J. Vol.10 (1967) p 135
- (KUCK68) Kuck, D.J. ILLIAC IV Software and application programming. IEEE Trans. on Computers Vol.C-17 (1968) p 758
- (MCIL60) McIlroy, M.D. Macro instruction extension of compiler language. Comm.ACM. Vol.3 (1960) p 214
- (MUTHU69) Muthukrishnan, C.R. Analysis and conversion of decision tables to computer programs. Doctoral Thesis (1969) Indian Institute of Technology, Kanpur.

- (REEV67) Reeves, C.M. Description of a syntax-directed translator. Comput.J. Vol.10 (1967) p 244
- (REIN66) Reinwald, L.T. and Soland, R.M. Conversion of limited entry decision tables to optimal computer programs I : Minimum average processing time. J.ACM. Vol.13 (1966) p 339
- (ROSE64) Rosen, S. A compiler-building system developed by Brooker and Morris. Comm.ACM. Vol.7 (1964) p 403

APPENDIX A

LISTING OF THE LITTLETRAN COMPILER

/IBFTC INPUT NOPRNT

SUBROUTINE INPUT

DATA BLAN/6H /

BLANK=BLAN

INTEGER WORD(72),PREV,CURRENT,BLANK

COMMON /CONNIE/ WORD ,PREV,CURRENT,INDEX,BLANK

READ 5,(WORD(I),I=1,72)

5 FORMAT(72A1)

K=7

DO 15 I=1,72

IF(I.LT.7)GO TO 15

IF(WORD(I).EQ.BLANK)GO TO 15

WORD(K)=WORD(I)

K=K+1

15 CONTINUE

WORD(K)=BLANK

PRINT 10,(WORD(I),I=1,K)

10 FORMAT(10X,72A1)

INDEX=7

RETURN

END

/IBFTC SCAN NOPRNT

SUBROUTINE SCAN

INTEGER WORD(72),PREV,CURRENT,BLANK

COMMON /CONNIE/ WORD ,PREV,CURRENT,INDEX,BLANK

IF(INDEX.EQ.1)GO TO 10

PREV=WORD(INDEX-1)

5 CURRENT=WORD(INDEX)

INDEX=INDEX+1

RETURN

10 PREV=BLANK

GO TO 5

END

/IBFTC CHAR NOPRNT

SUBROUTINE CHAR

INTEGER WORD(72),PREV,CURRENT,BLANK

COMMON /CONNIE/ WORD ,PREV,CURRENT,INDEX,BLANK

COMMON /JOSIE/ IFLAG1,IFLAG2,IFLAG3,IFLAG4

INTEGER CK1,CK2,CK3,CK4,CK5,CK6

DATA CK1,CK2,CK3,CK4,CK5,CK6/6H0 A,6H0 I,6H0 J,6H0
1R,6H0 S,6H0 Z/

KTEMP=CURRENT

KZERO=0

WRITE(99,10)KZERO,CURRENT

10 FORMAT(A1,4X,A1)

READ(99,15)CURRENT

15 FORMAT(A6)

IFLAG1=0

APPENDIX A

LISTING OF THE LITTLETRAN COMPILER

```

/IBFTC INPUT  NOPRNT
  SUBROUTINE INPUT
    DATA BLAN/6H      /
    BLANK=BLAN
    INTEGER              WORD(72),PREV,CURRENT,BLANK
    COMMON /CONNIE/ WORD    ,PREV,CURRENT,INDEX,BLANK
    READ 5,(WORD(I),I=1,72)
5    FORMAT(72A1)
    K=7
    DO 15 I=1,72
    IF(I.LT.7)GO TO 15
    IF(WORD(I).EQ.BLANK)GO TO 15
    WORD(K)=WORD(I)
    K=K+1
15   CONTINUE
    WORD(K)=BLANK
    PRINT 10,(WORD(I),I=1,K)
10   FORMAT(10X,72A1)
    INDEX=7
    RETURN
    END

/IBFTC SCAN  NOPRNT
  SUBROUTINE SCAN
    INTEGER              WORD(72),PREV,CURRENT,BLANK
    COMMON /CONNIE/ WORD    ,PREV,CURRENT,INDEX,BLANK
    IF(INDEX.EQ.1)GO TO 10
    PREV=WORD(INDEX-1)
5    CURRENT=WORD(INDEX)
    INDEX=INDEX+1
    RETURN
10   PREV=BLANK
    GO TO 5
    END

/IBFTC CHAR  NOPRNT
  SUBROUTINE CHAR
    INTEGER              WORD(72),PREV,CURRENT,BLANK
    COMMON /CONNIE/ WORD    ,PREV,CURRENT,INDEX,BLANK
    COMMON /JOSIE/ IFLAG1,IFLAG2,IFLAG3,IFLAG4
    INTEGER CK1,CK2,CK3,CK4,CK5,CK6
    DATA CK1,CK2,CK3,CK4,CK5,CK6/6H0    A,6H0    I,6H0    J,6H0
    1R,6H0    S,6H0    Z/
    KTEMP=CURRENT
    KZERO=0
    WRITE(99,10)KZERO,CURRENT
10   FORMAT(A1,4X,A1)
    READ(99,15)CURRENT
15   FORMAT(A6)
    IFLAG1=0

```

	EQ.1	EQ.0	EQ.1	EQ.0
IFLAG2				
N			GE.1	
ICAR				
CALL NUMBER	N.	N.	N.	X.
N=N+1	X.	X.	N.	N.
TEMP(N)=C	N.	N.	N.	N.
CALL SCAN	X.	X.	N.	N.
IFLAG2=2	N.	N.	N.	N.
CALL CHAR	X.	X.	N.	N.
CALL DIGIT	X.	X.	N.	N.
ICAR	N.	N.	N.	N.
PRINT 15	X.	X.	N.	N.
IFLAG3=1	N.	N.	X.	N.
GO TO 10	X.	X.	N.	N.
RETURN	N.	N.	N.	X.
N	GT.6	GT.6	GT.6	
IFLAG1	EQ.0	EQ.1		
IFLAG2	EQ.0		EQ.1	

ICAR			
CALL NUMBER	N.	N.	N.
N=N+1	N.	N.	N.
TEMP(N)=C	N.	N.	N.
CALL SCAN	N.	X.	X.
IFLAG2=2	N.	N.	N.
CALL CHAR	N.	X.	X.
CALL DIGIT	N.	X.	X.
ICAR	N.	N.	N.
PRINT 15	N.	N.	N.
IFLAG3=1	X.	N.	N.
GO TO 10	N.	X.	X.
RETURN	N.	N.	N.

*TLEND

15 FORMAT(1X,*ERROR- VARIABLE NAME HAS MORE THAN SIX CHARACTER

IF(N.GT.6)N=6

WRITE(99,20)

20

FORMAT(6X)

WRITE(99,25)(TEMP(I),I=1,N)

25

FORMAT(6A1)

READ(99,30)VNAME

30

FORMAT(A6)

RETURN

END

/IBFTC ARITH

SUBROUTINE ARITH

INTEGER

WORD(72),PREV,CURRENT,BLANK

INTEGER VNAME

COMMON /CONNIE/ WORD ,PREV,CURRENT,INDEX,BLANK

COMMON /JOSIE/ IFLAG1,IFLAG2,IFLAG3,IFLAG4

COMMON /SHEELA/VNAME

INTEGER RTPAR,EQSIGN,PLUS,DVD,PERIOD,EXPO

DATA LFPAR,RTPAR,EQSIGN,PLUS,MINUS,MPY,DVD,PERIOD,EXPO/

1

6H(,6H)

,6H=

,6H+

,6H-

,6H*

,6H/

2

6H. ,6H'

/

DATA NEG/6H\$

/

INTEGER ASIGN

NT=0

IF(IFLAG5.EQ.1)GO TO 13

```

        CALL SCAN
        CALL IDFIER
10      CONTINUE
*TABLE      TABLE FOR CHECKING FOR ASSIGNMENT STATEMENT
*THEAD
SXDT02      NC=02,NR=01,NA=01,RDEBUG,STACOL
IFLAG3      .EQ.1
CURENT      .EQ.EQSIGN
ASIGN=VNAME .X.
*ELSE
        IFLAG4=0
        INDEX=7
        RETURN
*TLEND
13      CALL LIST(MAIN)
        CALL LIST(WORKIN)
15      IFLAG3=2
17      CONTINUE
*TABLE      TABLE FOR ISOLATING OPERANDS IN ARITHMETIC EXPRESSIONS
*THEAD
SXDT03      NC=02,NR=05,NA=06,RDEBUG,STACOL
IFLAG3      .EQ.2      .EQ.1      .EQ.0      .EQ.0      .EQ.1
CURENT      .NE.BLANK  .EQ.BLANK  .NE.BLANK  .EQ.BLANK
CALL SCAN   .X.        .N.        .N.        .N.        .N.
CALL IDFIER .X.        .N.        .N.        .N.        .N.
CALL NEWTOP .N.        (VNAME,MAIN).N.      .N.        (VNAME,MAI
GO TO 18    .N.        .X.        .N.        .X.        .N.
GO TO 30    .N.        .N.        .X.        .N.        .X.
GO TO 17    .X.        .N.        .N.        .N.        .N.
*TLEND
18      IF(IFLAG3.EQ.0)GO TO 20
        KZERO=0
        WRITE(99,1)KZERO,VNAME
1      FORMAT(A1,4X,A1)
        READ(99,2)MODE
2      FORMAT(A6)
        INTEGER CK22
        DATA CK22/6H0      N/
        INTEGER CK1,CK2,CK3,CK4,CK5,CK6
        DATA CK1,CK2,CK3,CK4,CK5,CK6/6H0      A,6H0      I,6H0      J,6H0
1R,6H0      S,6H0      Z/
        IF(MODE.GE.CK2.AND.MODE.LE.CK22)GO TO 19
        IREAL=1
        GO TO 20
19      INTG=1
20      CONTINUE
*TABLE      TABLE FOR CONVERTING FROM INFIX TO SUFFIX POLISH
*TEXPR
        LOGICAL      BOOL1,BOOL2,BOOL3,BOOL4,BOOL5,BOOL6
        INTEGER C,TOPS
        BOOL1=.FALSE.
        BOOL2=.FALSE.
        BOOL3=.FALSE.
        BOOL4=.FALSE.
        BOOL5=.FALSE.
        BOOL6=.FALSE.

```

```

MTY=LISTMT(WORKIN)
TOPS=INTGER(TOP(WORKIN))
KFLAG=1
C=CURRENT
IF(TOPS.EQ.MPY.OR.TOPS.EQ.DVD)BOOL1=.TRUE.
IF(C.EQ.PLUS.OR.C.EQ.MINUS)BOOL2=.TRUE.
IF(C.EQ.MPY.OR.C.EQ.DVD)BOOL3=.TRUE.
IF(TOPS.EQ.PLUS.OR.TOPS.EQ.MINUS)BOOL4=.TRUE.
IF(C.EQ.EXPO.AND.TOPS.EQ.EXPO)BOOL5=.TRUE.
IF(PREV.NE.EQSIGN)BOOL6=.TRUE.

```

*THEAD

SXDT04

NC=10,NR=18,NA=08,RDEBUG,STACOL

MTY

.EQ.0

.NE.0

CURRENT

.NE.RTPAR

.NE.RTPAR

.EQ.MINUS

.EQ.MINUS

PREV

.NE.LFPAR

.EQ.EQSIGN

.EQ.LFPAR

BOOL1

.Y.

BOOL2

.Y.

TOPS

.EQ.LFPAR

BOOL3

BOOL4

BOOL5

BOOL6

.Y.

CALL NEWTOP (C,WORKIN)

(C,WORKIN)

.N.

.N.

.N.

CALL NEWTOP

.N.

.N.

.N.

.N.

(TOPS,

CALL POPTOP

.N.

.N.

.N.

.N.

(WORKI

PRINT 25

.N.

.N.

.N.

.N.

.N.

CURRENT=NEG

.N.

.N.

.X.

.X.

.N.

GO TO 15

.X.

.X.

.N.

.N.

.N.

GO TO 20

.N.

.N.

.X.

.X.

.N.

GO TO 40

.N.

.N.

.N.

.N.

.X.

MTY

.NE.0

.NE.0

.EQ.0

CURRENT

.EQ.RTPAR

.EQ.RTPAR

.EQ.RT

PREV

BOOL1

BOOL2

.Y.

TOPS

.EQ.NEG

.EQ.EXPO

.NE.LFPAR

.EQ.LFPAR

BOOL3

.Y.

BOOL4

BOOL5

BOOL6

CALL NEWTOP

.N.

.N.

.N.

.N.

.N.

CALL NEWTOP (TOPS,MAIN)

(TOPS,MAIN)

(TOPS,MAIN)

.N.

.N.

CALL POPTOP (WORKIN)

(WORKIN)

(WORKIN)

(WORKIN)

.N.

PRINT 25

.N.

.N.

.N.

.N.

.X.

CURRENT=NEG

.N.

.N.

.N.

.N.

.N.

GO TO 15

.N.

.N.

.N.

.X.

.X.

GO TO 20

.N.

.N.

.N.

.N.

.N.

GO TO 40

.X.

.X.

.X.

.N.

.N.

MTY

CURRENT

.EQ.LFPAR

.EQ.NEG

.EQ.NEG

PREV

BOOL1

.Y.

.Y.

BOOL2

.Y.

TOPS

.EQ.EXPO

.EQ.EXPO

BOOL3

.Y.

BOOL4

BOOL5

BOOL6

```

CALL NEWTOP (C,WORKIN)  .N.      .N.      .N.      .N.
CALL NEWTOP  .N.      (TOPS,MAIN) (TOPS,MAIN) (TOPS,MAIN) (TOPS,MAIN)
CALL POPTOP  .N.      (WORKIN)   (WORKIN)   (WORKIN)   (WORKIN)
PRINT 25     .N.      .N.      .N.      .N.      .N.
CURENT=NEG   .N.      .N.      .N.      .N.      .N.
GO TO 15     .X.      .N.      .N.      .N.      .N.
GO TO 20     .N.      .N.      .N.      .N.      .N.
GO TO 40     .N.      .X.      .X.      .X.      .X.
MTY
CURENT
PREV
BOOL1
BOOL2      .Y.
TOPS
      .EC.NEG
BOOL3
BOOL4      .Y.
BOOL5      .Y.
BOOL6
CALL NEWTOP  .N.      .N.      .N.
CALL NEWTOP (TOPS,MAIN) (TOPS,MAIN) (TOPS,MAIN)
CALL POPTOP (WORKIN)   (WORKIN)   (WORKIN)
PRINT 25     .N.      .N.      .N.
CURENT=NEG   .N.      .N.      .N.
GO TO 15     .N.      .N.      .N.
GO TO 20     .N.      .N.      .N.
GO TO 40     .X.      .X.      .X.
*ELSE
      CALL NEWTOP(C,WORKIN)
      GO TO 15
*TLEND
25      FORMAT(1X,*ERROR- ONE OR MORE UNBALANCED PARENTHESES*)
30      IF(LISTMT(WORKIN).NE.0)GO TO 35
      NRES=INTGER(POPTOP(MAIN))
      PRINT 51,NRES
      IF(IFLAG5.EQ.1)RETURN
      PRINT 60, ASIGN
      RETURN
35      IF(INTGER(TOP(WORKIN)).EQ.LFPAR)GO TO 37
      CALL NEWTOP(TOP(WORKIN),MAIN)
      CALL POPTOP(WORKIN)
      KFLAG=2
      GO TO 40
37      PRINT 25
      CALL POPTOP(WORKIN)
      GO TO 30
40      CONTINUE
      KSYMB=INTGER(POPTOP(MAIN))
      IF(KSYMB.EQ.NEG)GO TO 498
      IADR2=INTGER(POPTOP(MAIN))
      IADR1=INTGER(POPTOP(MAIN))
      GO TO 44
498      IADR2=INTGER(TOP(MAIN))
      DATA KT/4HTMP+/
74      FORMAT(1X,*ERROR- MIXED MODE IN ARITHMETIC EXPRESSION*)
80      FORMAT(A4,I2)
75      FORMAT(A4,I1,1X)
85      FORMAT(A6)
      INTEGER TMP
44      IF(NT.GT.9)GO 00 45

```

WRITE(99,75)KT,NT

READ(99,85)TMP

GO TO 47

45 WRITE(99,80) KT,NT

READ(99,85)TMP

47 IF(INTG.NE.1.OR.IREAL.NE.1)GO TO 50

IREAL=0

PRINT 74

50 CONTINUE

*TABLE

TABLE FOR GENERATING ORDER CODE FROM ARITHMETIC EXPRESSION

*THEAD

GMDT01

NC=03,NR=11,NA=20,RDEBUG,STACOL

INTG.EQ.1

.Y.

.Y.

.Y.

IREAL.EQ.1

.Y.

.Y.

KSYSB

.EQ.PLUS

.EQ.PLUS

.EQ.MINUS

.EQ.MINUS

.EQ.MPY

PRINT 51,

IADR1

IADR1

IADR1

IADR1

.N.

PRINT 52,

IADR2

.N.

.N.

.N.

.N.

PRINT 62,

.N.

IADR2

.N.

.N.

.N.

PRINT 53,

.N.

.N.

IADR2

.N.

.N.

PRINT 63,

.N.

.N.

.N.

IADR2

.N.

PRINT 54,

.N.

.N.

.N.

.N.

.N.

PRINT 55,

.N.

.N.

.N.

.N.

IADR1

PRINT 65,

.N.

.N.

.N.

.N.

IADR2

PRINT 56,

.N.

.N.

.N.

.N.

.N.

PRINT 54,

.N.

.N.

.N.

.N.

.N.

PRINT 57,

.N.

.N.

.N.

.N.

.N.

PRINT 67,

.N.

.N.

.N.

.N.

.N.

PRINT 58,

.N.

.N.

.N.

.N.

.N.

PRINT 54,

.N.

.N.

.N.

.N.

.N.

PRINT 55,

.N.

.N.

.N.

.N.

.N.

PRINT 65,

.N.

.N.

.N.

.N.

.N.

PRINT 66,

.N.

.N.

.N.

.N.

.N.

PRINT 59,

.N.

.N.

.N.

.N.

.N.

PRINT 60,

TMP

P

TMP

TMP

.N.

PRINT 70,

.N.

.N.

.N.

.N.

TMP

.Y.

.Y.

.Y.

.EQ.MPY

.EQ.DVD

.EQ.DVD

.EQ.EXPO

.EQ.EXPO

PRINT 51,

.N.

.N.

.N.

.N.

.N.

PRINT 52,

.N.

.N.

.N.

.N.

.N.

PRINT 62,

.N.

.N.

.N.

.N.

.N.

PRINT 53,

.N.

.N.

.N.

.N.

.N.

PRINT 63,

.N.

.N.

.N.

.N.

.N.

PRINT 54,

IADR1

.N.

.N.

.N.

.N.

PRINT 55,

.N.

.N.

.N.

.N.

.N.

PRINT 65,

IADR2

.N.

.N.

.N.

.N.

PRINT 56,

.N.

.X.

.X.

.N.

.N.

PRINT 54,

.N.

IADR1

IADR1

.N.

.N.

PRINT 57,

.N.

IADR2

.N.

.N.

.N.

PRINT 67,

.N.

.N.

IADR2

.N.

.N.

PRINT 58,

.N.

.N.

.N.

IADR2

IADR2

PRINT 54,

.N.

.N.

.N.

IADR1

IADR1

PRINT 55,

.N.

.N.

.N.

IADR1

.N.

PRINT 65,

.N.

.N.

.N.

.N.

IADR1

PRINT 66,

.N.

.N.

.N.

.N.

.N.

PRINT 59,

.N.

.N.

.N.

.X.

.X.

PRINT 60,

.N.

.N.

.N.

.N.

.N.

PRINT 70,

TMP

TMP

TMP

TMP

TMP

INTG.EQ.1
IREAL.EQ.1

KSYMB .EQ.NEG
PRINT 51, IADR2
PRINT 52, .N.
PRINT 62, .N.
PRINT 53, .N.
PRINT 63, .N.
PRINT 54, .N.
PRINT 55, .N.
PRINT 65, .N.
PRINT 56, .N.
PRINT 54, .N.
PRINT 57, .N.
PRINT 67, .N.
PRINT 58, .N.
PRINT 54, .N.
PRINT 55, .N.
PRINT 65, .N.
PRINT 66, .X.
PRINT 59, .N.
PRINT 60, IADR2
PRINT 70, .N.

*TLEND

51 FORMAT(10X,*CLA*,5X,A6)
52 FORMAT(10X,*ADD*,5X,A6)
53 FORMAT(10X,*SUB*,5X,A6)
54 FORMAT(10X,*LDQ*,5X,A6)
55 FORMAT(10X,*MPY*,5X,A6)
56 FORMAT(10X,*ZAC*)
57 FORMAT(10X,*DVP*,5X,A6)
58 FORMAT(10X,*AX0*,5X,A1,*-1,1*)
59 FORMAT(10X,*TIX*,5X,7H*-1,1,1)
60 FORMAT(10X,*STO*,5X,A6)
62 FORMAT(10X,*FAD*,5X,A6)
63 FORMAT(10X,*FSB*,5X,A6)
65 FORMAT(10X,*FMP*,5X,A6)
66 FORMAT(10X,*CHS*)
67 FORMAT(10X,*FDP*,5X,A6)
70 FORMAT(10X,*STQ*,5X,A6)

IF(KSYMB.EQ.NEG)GO TO 100

NT=NT+1

CALL NEWTOP(TMP,MAIN)

100 GO TO (20,30),KFLAG

END

/IBFTC NUMBER NOPRNO

SUBROUTINE NUMBER

INTEGER TEMP(5),PERIOD

N=0

INTEGER WORD(72),PREV,CURRENT,BLANK

INTEGER VNAME

COMMON /CONNIE/ WORD ,PREV,CURRENT,INDEX,BLANK

COMMON /JOSIE/ IFLAG1,IFLAG2,IFLAG3,IFLAG4

COMMON /SHEELA/VNAME

DATA PERIOD/6H. /

DATA IPOWER/6H' /

IEXP=0

IF(PREV.EQ.IPOWER)IEXP=1

```

10      CONTINUE
*TABLE      TABLE FOR RECOGNIZING DECIMAL NUMBERS IN ARITHMETIC EXPR
*TEXPR
      INTEGER C
      C=CURRENT
*THEAD
SXDT05      NC=03,NR=04,NA=08,RDEBUG,STACOL
IFLAG1      .EQ.1      .EQ.0      .EQ.0
IFLAG2      .EQ.1      .EQ.0      .EQ.0
CURRENT      .EQ.PERIOD      .NE.PERIOD
N=N+1      .X.      .N.      .X.      .N.
TEMP(N)=C      .X.      .N.      .X.      .N.
CALL SCAN      .X.      .X.      .X.      .N.
CALL CHAR      .X.      .X.      .X.      .N.
CALL DIGIT      .X.      .X.      .X.      .N.
PRINT 15      .N.      .X.      .N.      .N.
IFLAG3=1      .N.      .N.      .N.      .X.
GO TO 10      .X.      .X.      .X.      .N.
*TLEND
15      FORMAT(1X,*ERROR- ILLEGAL CHARACTER IN DECIMAL NUMBER*)
WRITE(99,20)
20      FORMAT(7X)
WRITE(99,25)(TEMP(I),I=1,N)
25      FORMAT(1H=,5A1)
READ(99,30)VNAME
30      FORMAT(A6)
IF(IEXP.EQ.1)READ(99,35)VNAME
35      FORMAT(1X,A6)
RETURN
END
/IBFTC IF.ART
SUBROUTINE IFART
      INTEGER      WORD(72),PREV,CURRENT,BLANK
      COMMON /CONNIE. WORD      ,PREV,CURRENT,INDEX,BLANK
      COMMON /JOSIE/ IFLAG1,IFLAG2,IFLAG3,IFLAG4
      DIMENSION S(6),T(6),STNO(3)
      DATA LFPAR,RTPAR,COMMA/6H(      ,6H)      ,6H,      /
      INTEGER RTPAR,COMMA,S,T,STNO
      IFLAG5=0
      DATA S/6H200000,6H020000,6H002000,6H000200,6H000020,6H000002/
      CALL SCAN
      INITA =0
*TABLE      TABLE FOR RECOGNIZING ARITHMETIC 'IF' STATEMENTS
*THEAD
SXDT06      NC=04,NR=03,NA=08,RDEBUG,STACOL
CURRENT      .EQ.F      .NE.LFPAR
PREV      .EQ.I      .EQ.F      .EQ.RTPAR
IFLAG2      .EQ.1
INITA      .EQ.0      .EQ.0
WORD(INDEX      .N.      .N.      -2)=BLANK
IFLAG5=1      .N.      .N.      .X.
INDEX=9      .N.      .N.      .X.
CALL AFITH      .N.      .N.      .X.
CALL SCAN      .X.      .N.      .X.
CALL DIGIT      .N.      .N.      .X.
GO TO 10      .X.      .N.      .N.
RETURN      .N.      .X.      .N.

```

*ELSE

INITA=1
CALL SCAN
CALL DIGIT
GO TO 10

*TLEND

N=1

14 WRITE(99,12)

12 FORMAT(6X).

*TABLE TABLE FOR THE CODE GENERATION OF ARITHMETIC 'IF' STATEMENT

*THEAD

SMDT02 NC=03,NR=03,NA=10,RDEBUG,STACOL

CURRENT .EQ.COMMA .EQ.BLANK .NE.COMMA

IFLAG2 .EQ.0

CURRENT .NF.BLANK

CALL SCAN .X. .N. .X.

CALL DIGIT .X. .N. .X.

WRITE(99,20)(T(I),I=1,K).N. .N.

READ(99,25) STNO(N) .N. .N.

N=N+1 .X. .N. .N.

PRINT 19 .N. .N. .X.

K=0 .X. .N. .N.

GO TO 30 .N. .X. .N.

NO TO 15 .N. .N. .X.

GO TO 14 .X. .N. .N.

*ELSE

K=K+1

1(K)=CURRENT

CALL SCAN

CALL DIGIT

GO TO 15

*TLEND

20 FORMAT(6A1)

25 FORMAT(A6)

30 DO 35 I=1,3

WRITE(99,31)STNO(I)

31 FORMAT(A6)

N=1

READ(99,32)(T(J),J=1,6)

32 FORMAT(6A1)

DO 33 M=1,6

IF(T(M).NE.BLANK)GO TO 33

GO TO 35

33 N=N+1

35 STNO(I)=OR(S(N),STNO(I))

PRINT 36,STNO(1)

36 FORMAT(10X,*TMI*,5X,A6)

PRINT 37,STNO(2)

37 FORMAT(10X,*TZE*,5X,A6)

PRINT 38,STNO(3)

38 FORMAT(10X,*TRA*,5X,A6)

RETURN

END

/IBFTC GOTO

SUBROUTINE GOTO

DATA LGOTO/6HGOTO /

INTEGER TEMP(6),COMMA,STNO(10),S(6)

DATA LFPAR,COMMA/6H

DATA LFPAR,RTPAR,COMMA/6H(,6H, ,6H, /

```

DO 10 I=1,4
CALL SCAN
10 TEMP(I)=CURENT
WRITE(99,15)(TEMP(I),I=1,4)
15 FORMAT(4A1)
READ(99,20)NAME
20 FORMAT(A4)
IFLAG6=0
IF(NAME.NE.LGOTO)RETURN
IFLAG6=1

*TABLE
*THEAD
SXDT05 NC=01,NA=01,NR=02
CURENT .EQ.COMMA .EQ.LFPAR
GO TO 30 .N. .X.
*ELSE
PRINT 21
21 FORMAT(1X,*ERROR=ILLEGAL 'GO TO' STATEMENT*)
RETURN

*TLEND
CALL SCAN
CALL DIGIT
CALL NUMBER
WRITE(99,22)VNAME
22 FORMAT(A6)
READ(99,23)(TEMP(I),I=1,6)
23 FORMAT(6A1)
N=1
DO 25 M=1,6
IF(TEMP(M).EQ.BLANK)GO TO 26
25 N=N+1
26 VNAME=OR(S(N),VNAME)
PRINT 27,VNAME
27 FORMAT(10X,*TRA*,5',A6)
30 CALL DIGIT
N=0
40 CONTINUE

*TABLE
*THEAD
SMDT03 NC=02,NR=03,NA=07
PREV .EQ.LFPAR .EQ.COMMA .EQ.RTPAR
IFLAG2 .EQ.1 .EQ.1
CALL NUMBER .X. .X. .N.
N=N+1 .X. .X. .N.
STN(N)=VNAME .X. .X. .N.
CALL SCAN .X. .X. .N.
CALL DIGIT .X. .X. .N.
CALL IDIFIER .N. .N. .X.
GO TO 40 .X. .X. .N.
*TLEND
IF(IFLAG3.NE.0)GO TO 50

```

```

      PRINT 45
45  FORMAT(1X,*ERROR- ILLEGAL =ARIABLE NAME IN COMPUTED 'GO TO' ST
      1ENT*)
      RETURN
50  PRINT 55,VNAME
55  FORMAT(10X,*LAC*,5X,A6,*,1*)
      PRINT 60
60  FORMAT(10X,*TRA*,5X,3H*,1)
      DO 70 I=1,N
      K=1
      DO 65 M=1,6
      IF(TEMP(M).EQ.BLANK)GO TO 70
65  K=K+1
70  STNO(I)=OR(S(K),STNO(I))
      DO 75 I=1,N
75  PRINT 80,STNO(I)
80  FORMAT(10X,*TRA*,5X,A6)
      RETURN
      END
/ENTRY

```

TO BE ISSUED 2103 4 APR 1970

[illegible]